# 1

# GIỚI THIỆU

## 1.1 MỞ ĐẦU

Vào năm 1971 tập đoàn Intel đã giới thiệu 8080, bộ vi xử lý ( microprocessor ) thành công đầu tiên. Sau đó không lâu Motorola, RCA, kế đến là MOS Technology và Zilog đã giới thiệu các bộ vi xử lý tương tự : 6800, 1801, 6502 và Z80. Bản thân các vi mạch ( IC : integrated circuit ) này tuy không có nhiều hiệu quả sử dụng nhưng khi là một phần của một máy tính đơn *board* ( single-board computer ), chúng trở thành thành phần trung tâm trong các sản phẩm có ích dùng để nghiên cứu và thiết kế. Các máy tính đơn *board* này, trong đó có *D2* của Motorola, *KIM-1* của MOS Technology và *SDK-85* của Intel là đáng ghi nhớ nhất, đã nhanh chóng xâm nhập vào các phòng thí nghiệm thiết kế của trường trung học, trường đại học và các công ty điện tử.

Vào năm 1976 Intel giới thiệu bộ vi điều khiển ( microcontroller ) 8748, một *chip* tương tự như các bộ vi xử lý và là *chip* đầu tiên trong họ vi điều khiển MCS-48. 8748 là một vi mạch chứa trên 17000 transistor bao gồm một CPU, 1 K byte EPROM, 64 byte RAM, 27 chân xuất nhập và một bộ định thời 8-bit. IC này và các IC khác tiếp theo của họ MCS-48 đã nhanh chóng trở thành chuẩn công nghiệp trong các ứng dụng hướng điều khiển ( control-oriented application ). Việc thay thế các thành phần cơ điện trong các sản phẩm như các máy giặt và các bộ điều khiển đèn giao thông là một ứng dụng phổ biến ban đầu. Các sản phẩm khác mà trong đó bộ vi điều khiển được tìm thấy bao gồm xe ô tô, thiết bị công nghiệp, các sản phẩm tiêu dùng và các ngoại vi của máy tính ( bàn phím của IBM-PC là một thí dụ sử dụng bộ vi điều khiển trong các thiết kế tối thiểu thành phần ).

Độ phức tạp, kích thước và khả năng của các bộ vi điều khiển được tăng thêm một bậc quan trọng vào năm 1980 khi Intel công bố *chip* 8051, bộ vi điều khiển đầu tiên của họ vi điều khiển MCS-51. So với 8048, *chip* 8051 chứa trên 60000 transistor bao gồm 4 K byte ROM, 128

byte RAM, 32 đường xuất nhập, 1 port nối tiếp và 2 bộ định thời 16-bit – một số lượng mạch đáng chú ý trong một IC đơn. Các thành viên mới được thêm vào cho họ MCS-51 và các biến thể ngày nay gần như có gấp đôi các đặc trưng này. Tập đoàn Siemens, nguồn sản xuất thứ hai các bộ vi điều khiển thuộc họ MCS-51 cung cấp *chip* SAB80515, một cải tiến của 8051 chứa trong một vỏ 68 chân, có 6 port xuất nhập 8-bit, 13 nguồn tạo ra ngắt và một bộ biến đổi A/D 8-bit với 8 kênh ngõ vào. Họ 8051 là một trong những bộ vi điều khiển 8-bit mạnh và linh hoạt nhất, đã trở thành bộ vi điều khiển hàng đầu trong những năm gần đây.

Quyển sách này khảo sát họ vi điều khiển MCS-51. Các chương tiếp theo giới thiệu cấu trúc phần cứng và phần mềm của họ MCS-51, đồng thời chứng minh qua nhiều thí dụ thiết kế, cách mà họ vi điều khiển này có thể tham gia vào các thiết kế điện tử với số thành phần thêm vào tối thiểu.

Trong các mục sau của chương này, thông qua việc giới thiệu vắt tắt về cấu trúc máy tính, ta sẽ phát triển từ vựng của nhiều từ được cấu tạo từ chữ đầu của các từ khác và các từ đang được sử dụng phổ biến nhưng dễ nhầm lẫn trong lĩnh vực này. Do nhiều thuật ngữ có định nghĩa mơ hồ và trùng lấp phụ thuộc vào định kiến của các tập đoàn lớn và các ý chợt nảy ra của nhiều tác giả khác nhau, cách giải quyết của chúng ta mang tính thực tiễn hơn là trừu tượng. Mỗi một thuật ngữ được giới thiệu trong khung cảnh chung nhất cùng với sự giải thích rõ ràng.

## 1.2 THUẬT NGỮ

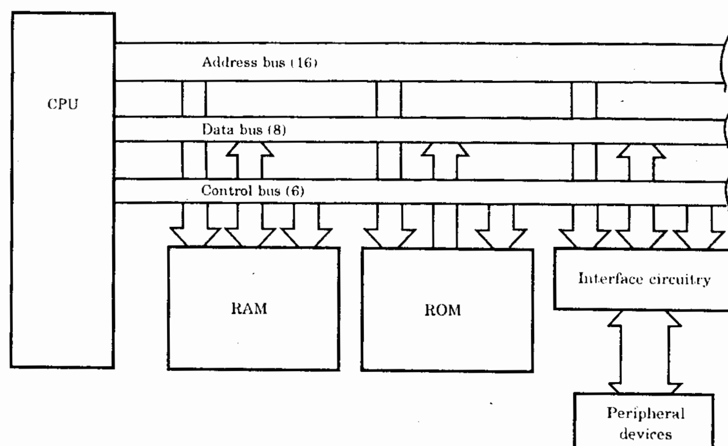
Một máy tính ( computer ) được định nghĩa bởi hai điểm chính :

- khả năng được lập trình để thao tác trên dữ liệu mà không cần đến sự can thiệp của con người.
- khả năng lưu trữ và khôi phục dữ liệu

Tổng quát hơn, một hệ máy tính ( computer system ) cũng bao gồm các thiết bị ngoại vi ( peripheral device ) để truyền thông với con người cũng như các chương trình ( program ) để xử lý dữ liệu. Thiết bị là phần cứng ( hardware ) và chương trình là phần mềm ( software ). Chúng ta hãy bắt đầu với phần cứng của máy tính bằng cách khảo sát hình 1.1.

Hình 1.1 là sơ đồ khối đơn giản, không chi tiết một cách cố ý nhằm mục đích tiêu biểu cho tất cả các loại máy tính. Như ta đã thấy, một hệ máy tính bao gồm một đơn vị xử lý trung tâm CPU ( central processing unit ), đơn vị này kết nối với bộ nhớ truy xuất ngẫu nhiên RAM ( random access memory ) và bộ nhớ chỉ đọc ROM ( read only memory ) thông qua bus địa chỉ ( address bus ), bus dữ liệu ( data bus ) và bus điều

khiển ( control bus ). Các mạch giao tiếp ( interface circuit ) kết nối các bus của hệ thống ( gọi tắt là bus hệ thống ) với các thiết bị ngoại vi. Ta sẽ đề cập chi tiết các đơn vị vừa nêu trên.




Hình 1.1 : Sơ đồ khối của một hệ máy vi tính

CPU : đơn vị xử lý trung tâm

RAM : bộ nhớ truy xuất ngẫu nhiên ( hay bộ nhớ đọc / ghi )

ROM : bộ nhớ chỉ đọc

Interface circuitry : mạch giao tiếp

Peripheral devices : các thiết bị ngoại vi

Address bus : bus địa chỉ

Data bus : bus dữ liệu

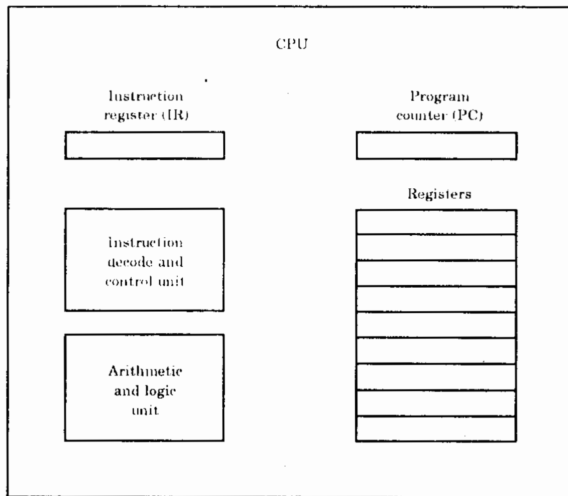
Control bus : bus điều khiển

## 1.3 ĐƠN VỊ XỬ LÝ TRUNG TÂM

CPU, trái tim của hệ máy tính, quản lý tất cả các hoạt động của hệ và thực hiện tất cả các thao tác trên dữ liệu. Hầu hết các CPU chỉ bao gồm một tập các mạch logic thực hiện liên tục hai thao tác : tìm nạp lệnh và thực thi lệnh. CPU có khả năng hiểu và thực thi các lệnh dựa trên một tập các mã nhị phân, mỗi một mã nhị phân biểu thị một thao tác đơn giản. Các lệnh này thường là các lệnh số học ( như cộng, trừ, nhân, chia ), các lệnh logic ( như AND, OR, NOT, v.v... ), các lệnh di chuyển dữ liệu hoặc các lệnh rẽ nhánh, được biểu thị bởi một tập các mã nhị phân và được gọi là tập lệnh ( instruction set ).

Hình 1.2 cho ta một cái nhìn rất đơn giản bên trong của CPU. Hình này trình bày một tập các thanh ghi ( register ) có nhiệm vụ lưu giữ tạm thời các thông tin, một đơn vị số học logic ALU ( arithmetic-logic unit )

có nhiệm vụ thực hiện các thao tác trên các thông tin này, một đơn vị giải mã lệnh và điều khiển ( instruction decode and control unit ) có nhiệm vụ xác định thao tác cần thực hiện và thiết lập các hoạt động cần thiết để thực hiện thao tác. CPU còn có hai thanh ghi nữa : thanh ghi lệnh IR ( instruction register ) lưu giữ mã nhị phân của lệnh để được thực thi và bộ đếm chương trình PC ( program counter ) lưu giữ địa chỉ của lệnh kế tiếp trong bộ nhớ cần được thực thi.




Hình 1.2 : Đơn vị xử lý trung tâm CPU

CPU : đơn vị xử lý trung tâm

Instruction register ( IR ) : thanh ghi lệnh ( IR )

Instruction decode and control unit : đơn vị giải mã lệnh và điều khiển

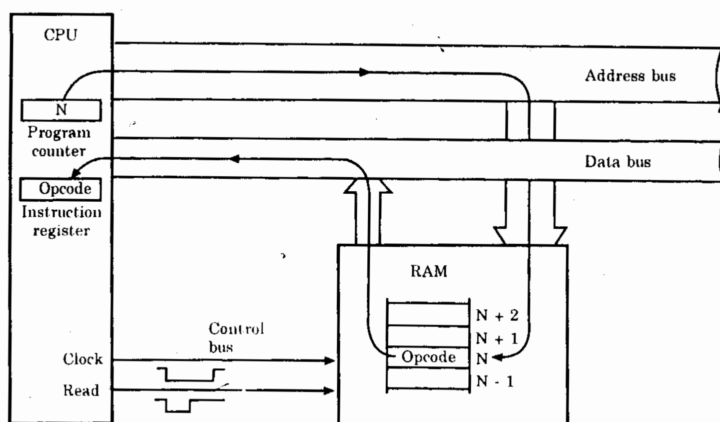
Program counter ( PC ) : bộ đếm chương trình ( PC )

Registers : các thanh ghi

Việc tìm nạp một lệnh từ RAM hệ thống là một trong các thao tác cơ bản nhất mà CPU thực hiện. Việc tìm nạp lệnh được thực hiện theo các bước sau :

- nội dung của PC được đặt lên bus địa chỉ
- tín hiệu điều khiển READ được xác lập ( chuyển sang trạng thái tích cực )
- dữ liệu ( opcode của lệnh ) được đọc từ RAM và đưa lên bus dữ liệu
- opcode được chốt vào thanh ghi lệnh bên trong CPU
- PC được tăng để chuẩn bị tìm nạp lệnh kế từ bộ nhớ

Hình 1.3 minh họa luồng thông tin cho việc tìm nạp lệnh.




Hình 1.3 : Hoạt động của bus cho chu kỳ tìm nạp lệnh

CPU : đơn vị xử lý trung tâm
Program counter : bộ đếm chương trình
Instruction register : thanh ghi lệnh
Clock : chân xung *clock*
Read : chân điều khiển đọc
Address bus : bus địa chỉ
Data bus : bus dữ liệu
Control bus : bus điều khiển

Giai đoạn thực thi lệnh bao gồm việc giải mã opcode và tạo ra các tín hiệu điều khiển, các tín hiệu này điều khiển việc xuất nhập giữa các thanh ghi nội với ALU và thông báo để ALU thực hiện thao tác đã được xác định. Do các thao tác có tầm thay đổi rộng, phạm vi dành cho các giải thích vừa nêu trên có phần nào bị giới hạn, chỉ áp dụng được cho các thao tác đơn giản như tăng nội dung của một thanh ghi. Các lệnh phức tạp hơn đòi hỏi thêm nhiều bước nữa, chẳng hạn như đọc byte dữ liệu thứ hai và byte dữ liệu thứ ba để thực hiện thao tác.

Một chuỗi các lệnh được kết hợp để thực hiện một công việc có ý nghĩa được gọi là một chương trình ( program ) hay phần mềm ( software ). Mức độ mà những công việc được thực hiện đúng và có hiệu quả phần lớn được xác định bởi chất lượng của phần mềm, không phải bởi sự phức tạp của CPU. Vậy thì các chương trình " điều khiển " CPU trong khi làm việc đôi khi dẫn đến sai nhầm, chính là do những nhược điểm của các tác giả chương trình. Các câu như là " máy tính tạo ra một lỗi " là sai. Mặc dù thiết bị có sự cố là điều không thể tránh được, các lỗi được tạo ra thường là dấu hiệu của các chương trình tồi hoặc lỗi của người điều khiển.

## 1.4 BỘ NHỚ BÁN DẪN : RAM VÀ ROM

Các chương trình và dữ liệu được lưu giữ trong bộ nhớ. Các biến thể của bộ nhớ máy tính và các thành phần kèm theo rất đa dạng, và công nghệ thường bị đột phá nên việc nghiên cứu liên tục và bao quát bộ nhớ đòi hỏi phải theo kịp các phát triển mới nhất. Các bộ nhớ được truy xuất trực tiếp bởi CPU bao gồm các IC bán dẫn gọi là RAM và ROM. Có hai đặc trưng dùng để phân biệt RAM và ROM : thứ nhất RAM là bộ nhớ đọc / ghi trong khi ROM là bộ nhớ chỉ đọc, thứ hai RAM không tiếp tục lưu giữ nội dung khi bị mất nguồn cấp điện trong khi ROM thì ngược lại.

Hầu hết các hệ máy tính đều có ổ đĩa và một dung lượng ROM nhỏ chỉ cần đủ để lưu giữ các chương trình ngắn, thường sử dụng, nhằm thực hiện các thao tác xuất nhập. Các chương trình và dữ liệu của người sử dụng được lưu trên đĩa và được nạp vào RAM để thực thi. Với giá thành liên tục được giảm thấp, các hệ máy tính nhỏ thường chứa bộ nhớ RAM từ hàng triệu byte đến hàng trăm triệu byte.

## 1.5 CÁC BUS : ĐỊA CHỈ, DỮ LIỆU VÀ ĐIỀU KHIỂN

Một bus là một tập các dây mang thông tin có cùng một mục đích. Việc truy xuất tới một mạch xung quanh CPU sử dụng ba bus : bus địa chỉ, bus dữ liệu và bus điều khiển. Với mỗi thao tác đọc hoặc ghi, CPU xác định rõ vị trí của dữ liệu ( hoặc lệnh ) bằng cách đặt một địa chỉ lên bus địa chỉ, sau đó tích cực một tín hiệu trên bus điều khiển để chỉ ra thao tác là đọc hay ghi. Thao tác đọc lấy một byte dữ liệu từ bộ nhớ ở vị trí đã xác định và đặt byte này lên bus dữ liệu. CPU đọc dữ liệu và đặt dữ liệu vào một trong các thanh ghi nội của CPU. Với thao tác ghi, CPU xuất dữ liệu lên bus dữ liệu. Nhờ vào tín hiệu điều khiển, bộ nhớ nhận biết đây là thao tác ghi và lưu dữ liệu vào vị trí đã được xác định.

Hầu hết các máy tính nhỏ có từ 16 đến 32 đường địa chỉ và có khả năng truy xuất $2^n$ vị trí nhớ. Một bus địa chỉ 16-bit do vậy có thể truy xuất một bộ nhớ có 64 K vị trí nhớ, một bus địa chỉ 20-bit có khả năng truy xuất 1 M vị trí nhớ và một bus địa chỉ 32-bit có khả năng truy xuất đến 4 G vị trí nhớ ( 1 K = 1024, 1 M = 1024 K và 1 G = 1024 M ).

Bus dữ liệu mang thông tin giữa CPU và bộ nhớ cũng như giữa CPU và các thiết bị xuất nhập. Việc cố gắng nghiên cứu bao quát phần lớn được dùng vào việc xác định loại các hoạt động làm mất nhiều thời gian thực thi đáng giá của máy tính. Hiển nhiên là máy tính sử dụng đến 2/3 thời gian vào các việc di chuyển dữ liệu. Vì đa số các thao tác di chuyển dữ liệu xảy ra giữa một thanh ghi của CPU với ROM và RAM ngoài, số đường ( hay độ rộng ) của bus dữ liệu rất quan trọng đối với hiệu suất tổng thể của máy tính. Giới hạn bởi độ rộng này có dạng cổ chai : có thể

có một lượng rất lớn bộ nhớ trong hệ thống và CPU có thể có khả năng tính toán rất lớn nhưng việc truy xuất dữ liệu – di chuyển dữ liệu giữa bộ nhớ và CPU thông qua bus dữ liệu – thường bị nghẽn như cổ chai do độ rộng của bus dữ liệu.

Điều này rất quan trọng nên người ta thường thêm một tiền tố ( trong tiếng Anh ) để chỉ ra sự mở rộng để tránh hiện tượng cổ chai. Câu máy tính 16-bit chỉ ra rằng máy tính có 16 đường dữ liệu. Các máy tính được phân loại 4-bit, 8-bit, 16-bit, 32-bit có khả năng tính toán tổng thể tăng khi độ rộng của bus dữ liệu tăng.

Lưu ý là bus dữ liệu, như được chỉ ra trong hình 1.1, là bus hai chiều còn bus địa chỉ là bus một chiều. Các thông tin địa chỉ luôn luôn được cung cấp bởi CPU ( được chỉ bởi mũi tên trong hình 1.1 ) trong khi dữ liệu di chuyển theo cả hai hướng tùy thuộc vào thao tác được dự định là đọc hay ghi. Cũng lưu ý là thuật ngữ " dữ liệu " được sử dụng theo nghĩa tổng quát : " thông tin " di chuyển trên bus dữ liệu có thể là lệnh của chương trình, địa chỉ theo sau lệnh hoặc dữ liệu được sử dụng bởi chương trình.

Bus điều khiển là một hỗn hợp các tín hiệu, mỗi một tín hiệu có một vai trò riêng trong việc điều khiển có trật tự hoạt động của hệ thống. Theo lệ thường, các tín hiệu điều khiển là các tín hiệu định thời được cung cấp bởi CPU để đồng bộ việc di chuyển thông tin trên các bus địa chỉ và dữ liệu. Mặc dù thông thường có ba tín hiệu như là CLOCK, READ và WRITE đối với việc di chuyển dữ liệu cơ bản giữa CPU và bộ nhớ, tên và hoạt động của các tín hiệu điều khiển phụ thuộc nhiều vào CPU cụ thể. Ta cần nghiên cứu chi tiết các tham khảo kỹ thuật của các nhà sản xuất.

## 1.6 CÁC THIẾT BỊ XUẤT NHẬP

Các thiết bị xuất nhập hay các thiết bị ngoại vi của máy tính cho ta đường truyền thông giữa hệ máy tính với thế giới bên ngoài. Không có các thiết bị ngoại vi, các hệ máy tính chỉ là những chiếc máy bị thu hẹp và ít được sử dụng. Tổng quát có ba loại thiết bị xuất nhập là các thiết bị lưu trữ lớn, các thiết bị giao tiếp với con người và các thiết bị điều khiển / kiểm tra.

### 1.6.1 Các thiết bị lưu trữ lớn

Cũng như các bộ nhớ bán dẫn RAM và ROM, các thiết bị lưu trữ lớn luôn luôn tăng trưởng và phát triển. Như tên gọi, các thiết bị lưu trữ lớn lưu trữ các lượng lớn thông tin ( chương trình hoặc dữ liệu ) mà các thông tin này không thể chứa đủ trong RAM tương đối nhỏ ( còn gọi là bộ nhớ chính ) của máy tính. Thông tin này phải được nạp vào trong bộ

nhớ chính trước khi CPU truy xuất chúng. Nếu ta phân loại theo sự truy xuất, các thiết bị lưu trữ lớn hoặc thuộc loại *online* hoặc thuộc loại *archival*. Bộ lưu trữ loại *online* thường là đĩa từ thích hợp với CPU không có sự can thiệp của con người khi yêu cầu một chương trình, bộ lưu trữ *archival* thường là đĩa hoặc băng từ mặc dù các đĩa quang như là CD-ROM hoặc công nghệ WORM hiện đang được ưa chuộng và có thể thay thế các bộ lưu trữ *archival* do độ tin cậy, khả năng lưu trữ và giá thành thấp.

### 1.6.2 Các thiết bị giao tiếp với con người

Việc liên kết con người và máy được thực hiện qua nhiều thiết bị giao tiếp với con người mà thông thường nhất là thiết bị đầu cuối hiển thị *video* VDT ( video display terminal ) và máy in. Máy in là thiết bị xuất còn các VDT thực ra là 2 thiết bị vì chúng chứa một bàn phím để nhập và một đèn tia âm cực CRT ( cathode ray tube ) để xuất. Một lĩnh vực kỹ thuật, được gọi là " các nhân tố con người ", đã phát triển từ nhu cầu thiết kế các thiết bị ngoại vi cho con người với mục đích là an toàn, tiện nghi và hiệu quả cùng với các đặc tính của con người đối với những máy mà con người sử dụng. Từ đó ta thấy có nhiều công ty sản xuất ra các thiết bị ngoại vi hơn là các công ty sản xuất ra máy tính.

Đối với hầu hết các hệ máy tính, thường ta có tối thiểu 3 thiết bị : một bàn phím, một CRT và một máy in. Các thiết bị khác giao tiếp với con người bao gồm : cần điều khiển trò chơi, bút sáng, con chuột, ống nói, loa v.v... .

### 1.6.3 Các thiết bị điều khiển / kiểm tra

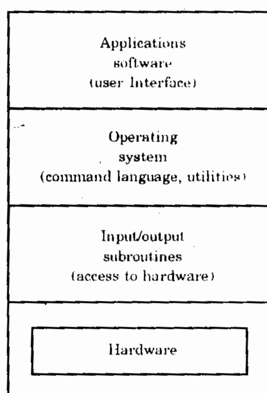
Nhờ vào các thiết bị điều khiển / kiểm tra, các máy tính có thể thực hiện vô số các tác vụ hướng điều khiển cũng như thực hiện chúng không ngơi nghỉ, không mệt mỏi và điều này vượt xa khả năng của con người. Nhiều ứng dụng trong đời sống hoặc trong công nghiệp sử dụng các thiết bị này.

Các thiết bị điều khiển là các thiết bị xuất hoặc các bộ kích thích ( actuator ), các thiết bị kiểm tra là các thiết bị nhập hoặc các cảm biến biến đổi các đại lượng phi điện như nhiệt, ánh sáng, áp suất, v.v... thành các đại lượng điện như điện áp hay dòng điện để máy tính đọc. Mạch giao tiếp biến đổi điện áp hay dòng điện này thành các mã nhị phân hoặc ngược lại và thông qua phần mềm, một quan hệ được thiết lập giữa các thiết bị nhập và các thiết bị xuất.

Việc giao tiếp bằng phần cứng và phần mềm giữa các thiết bị này với các bộ vi điều khiển là một trong các chủ đề chính của quyển sách này.

## 1.7 CHƯƠNG TRÌNH : LỚN VÀ NHỎ

Các thảo luận ở trên tập trung vào phần cứng của các hệ máy tính, các chương trình hoặc phần mềm điều khiển phần cứng làm việc chỉ mới đề cập qua. Tầm quan trọng tương đối của phần cứng so với phần mềm đã được dịch chuyển một cách rõ rệt trong thập niên cuối của thế kỷ 20. Trong khi trước đây giá sản xuất và bảo trì phần cứng của máy tính rất đắt so với giá thành của phần mềm, ngày nay với các *chip* có độ tích hợp cao LSI ( large scale integrated ) và rất cao VLSI ( very large scale integrated ) giá thành của phần cứng giảm đi rất nhiều. Các công việc đòi hỏi nghiên cứu sâu như viết, cung cấp và thu thập tư liệu, bảo trì, cập nhật và phân phối phần mềm chiếm phần lớn giá thành trong việc tự động hóa quá trình sử dụng các máy tính.




Hình 1.4 : Các cấp phần mềm

Applications software : phần mềm ứng dụng
User interface : giao diện với người sử dụng
Operating system : hệ điều hành
Command language, utilities : ngôn ngữ lệnh, các tiện ích
Input/output subroutines : các chương trình con xuất nhập
Access to hardware : truy xuất đến phần cứng
Hardware : phần cứng

Ta hãy khảo sát các loại phần mềm khác nhau. Hình 1.4 minh họa ba cấp phần mềm có vị trí ở giữa người sử dụng và phần cứng của một hệ máy tính : phần mềm ứng dụng ( application software ), hệ điều hành ( operating system ) và các chương trình con xuất nhập ( input/output subroutine ).

Ở cấp thấp nhất các chương trình con xuất nhập trực tiếp quản lý phần cứng của hệ thống, đọc các ký tự từ bàn phím, đưa các ký tự lên CRT, đọc một khối thông tin từ đĩa và v.v... Do các chương trình con này

liên kết mật thiết với phần cứng, chúng được viết bởi các chuyên gia thiết kế phần cứng và thường được lưu giữ trong ROM ( chúng là hệ xuất nhập cơ bản BIOS [ basic input output system ] trên máy tính cá nhân PC [ personal computer ] của IBM chẳng hạn ).

Để giúp cho người lập trình dễ dàng truy cập đến phần cứng của hệ thống, các điều kiện nhập và thoát được xác định một cách rõ ràng đối với các chương trình con xuất nhập. Người lập trình chỉ cần khởi động các giá trị cho các thanh ghi của CPU và gọi chương trình con : thao tác cần thiết được thực thi và kết quả sẽ trả về trong các thanh ghi của CPU hoặc lưu lại trong RAM hệ thống.

Để bổ sung đầy đủ cho các chương trình con xuất nhập, ROM chứa một chương trình khởi động ( start-up program ), chương trình này được thực thi khi hệ thống được cấp điện hoặc được thiết lập lại ( reset ) bằng tay. Bản chất không bị mất nội dung của ROM được sử dụng ở đây vì chương trình khởi động phải hiện hữu trong thời gian khởi động hệ thống. Chương trình này kiểm tra các tùy chọn, khởi động bộ nhớ, thực hiện việc kiểm tra chẩn đoán hư hỏng v.v... Cuối cùng nhưng không kém quan trọng, một trình nạp *bootstrap* ( bootstrap loader ) đọc *track* đầu tiên ( một **chương trình nhỏ** ) từ đĩa vào RAM và chuyển điều khiển tới chương trình nhỏ này để nạp phần thường trú trong RAM của hệ điều hành ( một **chương trình lớn** ) từ đĩa và chuyển điều khiển tới chương trình lớn này, hoàn tất việc khởi động hệ thống.

Hệ điều hành là một tập hợp lớn các chương trình được nạp vào trong một hệ máy tính nhằm cung cấp các cơ chế truy xuất, quản lý và sử dụng một cách có hiệu quả các tài nguyên ( resource ) của máy tính. Các khả năng này được thể hiện thông qua ngôn ngữ lệnh điều khiển ( command language ) và các chương trình tiện ích ( utility program ) của hệ điều hành, và đến lượt chúng tạo điều kiện thuận lợi cho việc phát triển các phần mềm ứng dụng. Nếu một phần mềm ứng dụng được thiết kế tốt, người sử dụng tác động tương hỗ với máy tính mà không cần có kiến thức nhiều về hệ điều hành. Cung cấp một giao diện với người sử dụng một cách an toàn, hiệu quả là một trong các mục tiêu cơ bản của việc thiết kế các phần mềm ứng dụng.

## 1.8 MICRO, MINI VÀ MAINFRAME

Chúng ta phân loại máy tính dựa theo độ lớn và khả năng tính toán : máy vi tính ( microcomputer ), máy tính *mini* ( minicomputer ) và máy tính *mainframe* ( mainframe computer ). Đặc điểm chính của máy vi tính là kích thước và khả năng đóng gói của CPU. Máy vi tính chứa bên trong một vi mạch tích hợp đơn gọi là bộ vi xử lý ( microprocessor ). Các máy tính *mini* và *mainframe* thì phức tạp hơn trong từng chi tiết cấu

trúc, chúng có các CPU chứa nhiều vi mạch tích hợp, từ vài IC đối với máy tính *mini* đến vài *board* chứa các IC đối với máy tính *mainframe*. Đây là điều cần thiết để có được tốc độ cao và khả năng tính toán mạnh.

Các máy vi tính điển hình của IBM-PC, Apple Macintosh và Commodore Amiga sử dụng một bộ vi xử lý làm CPU. RAM, ROM và các mạch giao tiếp yêu cầu nhiều vi mạch với số thành phần thường tăng theo khả năng tính toán. Các mạch giao tiếp có độ phức tạp thay đổi tùy thuộc vào các thiết bị xuất nhập. Chẳng hạn để điều khiển một loa chứa trong hầu hết các máy vi tính ta chỉ cần một cặp cổng logic, tuy nhiên mạch giao tiếp và điều khiển ổ đĩa cần nhiều vi mạch ở dạng LSI.

Có một đặc trưng khác phân biệt máy vi tính, máy tính *mini* và máy tính *mainframe*. Các máy vi tính là các hệ thống chỉ dành cho một người sử dụng và thuộc loại đơn tác vụ, chúng chỉ tác động tương hỗ với một người sử dụng và chúng chỉ thực thi một chương trình ở một thời điểm. Các máy tính *mini* và *mainframe* là các hệ thống dành cho nhiều người sử dụng và thuộc loại đa tác vụ, chúng có thể tác động tương hỗ với nhiều người sử dụng và thực thi đồng thời nhiều chương trình. Thực tế, việc thực thi đồng thời nhiều chương trình là ảo giác được tạo ra từ việc chia xẻ thời gian sử dụng các tài nguyên của CPU. Tuy nhiên, với các hệ đa xử lý có nhiều CPU, các tác vụ được thực thi đồng thời.

## 1.9 TỪ BỘ VI XỬ LÝ ĐẾN BỘ VI ĐIỀU KHIỂN

Như đã đề cập ở các phần trên, các bộ vi xử lý là các CPU đơn *chip* được sử dụng trong các máy vi tính. Vậy thì đâu là sự khác nhau giữa một bộ vi điều khiển và một bộ vi xử lý. Câu hỏi này có thể được trả lời từ 3 phối cảnh : cấu trúc phần cứng ( hardware architecture ), các ứng dụng và các đặc trưng của tập lệnh ( instruction set feature ).

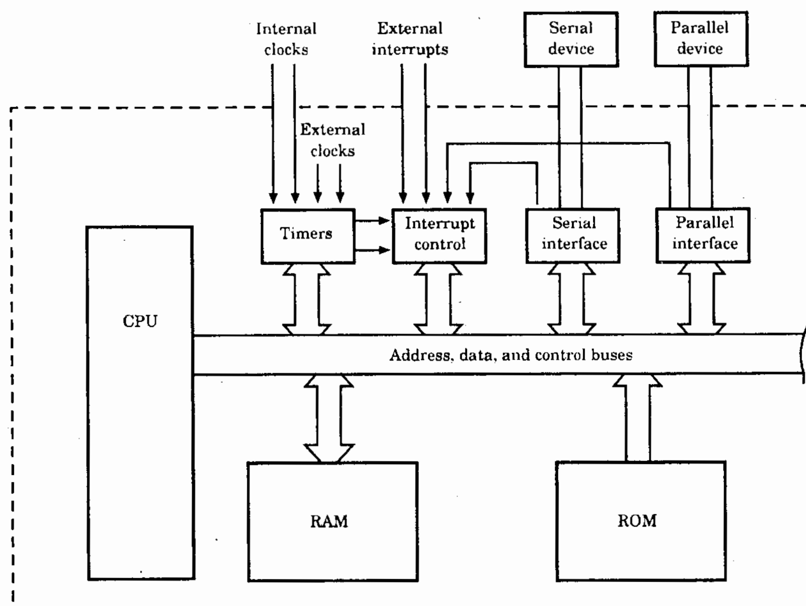
### 1.9.1 Cấu trúc phần cứng

Để chỉ rõ sự khác nhau giữa các bộ vi điều khiển và các bộ vi xử lý, hình 1.2 được vẽ lại chi tiết hơn ở hình 1.5.

Trong khi bộ vi xử lý là một CPU đơn *chip*, một bộ vi điều khiển là một vi mạch đơn chứa bên trong một CPU và các mạch khác để tạo nên một hệ máy vi tính đầy đủ. Các thành phần ở bên trong của khung vẽ bằng các đường không liên tục ở hình 1.5 là phần chủ yếu của hầu hết các bộ vi điều khiển.

Ngoài CPU, các bộ vi điều khiển còn chứa bên trong chúng các RAM, ROM, mạch giao tiếp nối tiếp, mạch giao tiếp song song, bộ định thời và các mạch điều khiển ngắt, tất cả hiện diện bên trong một vi mạch. Dĩ

nhiên dung lượng của RAM trong *chip* không thể đạt bằng với dung lượng RAM ở các máy vi tính nhưng như ta sẽ khảo sát sau, điều này không phải là một hạn chế vì các bộ vi điều khiển được thiết kế với dự định dành cho những ứng dụng hoàn toàn khác.




Hình 1.5 : Sơ đồ khối chi tiết của một hệ máy vi tính

CPU : đơn vị xử lý trung tâm

Timers : các bộ định thời

Interrupt control : điều khiển ngắt

Serial interface : giao tiếp nối tiếp

Parallel interface : giao tiếp song song

Address, data and control buses : các bus địa chỉ, dữ liệu và điều khiển

RAM : bộ nhớ đọc / ghi

ROM : bộ nhớ chỉ đọc

External clocks : các xung *clock* bên ngoài

External interrupts : các ngắt ngoài

Serial device : thiết bị nối tiếp

Parallel device : thiết bị song song

Một đặc trưng quan trọng của bộ vi điều khiển là hệ thống ngắt được thiết kế bên trong *chip*. Cũng như các thiết bị hướng điều khiển, các bộ vi điều khiển đáp ứng với các tác động bên ngoài ( các ngắt ) theo thời gian thực. Chúng phải thực hiện việc chuyển đổi ngữ cảnh rất nhanh,

treo một quá trình trong khi đang thực thi một quá trình khác theo yêu cầu của một sự kiện. Dĩ nhiên hầu hết các bộ vi xử lý đều có khả năng hiện thực các sơ đồ ngắt ( interrupt scheme ) nhưng phải sử dụng các thành phần bên ngoài trong khi đó mạch bên trong của một *chip* vi điều khiển bao gồm các mạch quản lý ngắt cần thiết.

### 1.9.2 Các ứng dụng

Các bộ vi xử lý hầu hết được dùng làm các CPU trong các hệ máy vi tính trong khi các bộ vi điều khiển được tìm thấy trong các thiết kế nhỏ, với số thành phần thêm vào tối thiểu nhằm thực hiện các hoạt động hướng điều khiển. Trong quá khứ các thiết kế như vậy yêu cầu hàng chục hoặc thậm chí hàng trăm vi mạch số. Bộ vi điều khiển giúp ta giảm thiểu số lượng tổng thể các thành phần. Tất cả chỉ cần một bộ vi điều khiển, một số ít các thành phần hỗ trợ và một chương trình điều khiển chứa trong ROM. Các bộ vi điều khiển thích hợp với các ứng dụng điều khiển thiết bị xuất nhập trong các thiết kế yêu cầu số thành phần tối thiểu, trong khi đó các bộ vi xử lý thích hợp với các ứng dụng xử lý thông tin trong các hệ máy tính.

### 1.9.3 Các đặc trưng của tập lệnh

Từ các khác nhau về ứng dụng, các bộ vi điều khiển có các yêu cầu khác đối với tập lệnh của chúng so với các bộ vi xử lý. Các tập lệnh của các bộ vi xử lý bao gồm các lệnh xử lý bao quát nên chúng mạnh về các kiểu định địa chỉ với các lệnh cung cấp các hoạt động trên các lượng dữ liệu lớn. Các lệnh của chúng có thể hoạt động trên các ½ byte, byte, từ, từ kép. Các kiểu định địa chỉ cung cấp khả năng truy xuất các dãy dữ liệu lớn bằng cách sử dụng các con trỏ địa chỉ và các *offset*. Các kiểu tăng và giảm tự động làm đơn giản hóa các bước thực thi trên các dãy dữ liệu ở các giới hạn byte, từ và từ kép. Các lệnh đặc quyền không thể thực thi trong các chương trình của người sử dụng và việc liệt kê còn tiếp tục nữa nếu ta muốn.

Các bộ vi điều khiển có các tập lệnh cung cấp các điều khiển xuất nhập. Mạch giao tiếp cho nhiều ngõ nhập và ngõ xuất chỉ sử dụng một bit. Thí dụ một động cơ có thể được điều khiển chạy hoặc dừng bằng cách cung cấp tín hiệu điều khiển từ một *port* 1-bit. Các bộ vi điều khiển có các lệnh *set* và xóa các bit đơn và thực thi các thao tác hướng bit ( bit oriented operation ) như là AND, OR, XOR, nhảy nếu bit được *set* hoặc được xóa, v.v...

Đặc trưng mạnh này hiếm khi thấy trong các bộ vi xử lý thường được thiết kế để hoạt động trên các byte hoặc các đơn vị dữ liệu lớn hơn.

Trong các thiết bị điều khiển và kiểm tra, các bộ vi điều khiển có các mạch bên trong và các lệnh dành cho các thao tác xuất nhập, định thời sự kiện, cho phép và thiết lập các mức ưu tiên cho các ngắt được tạo ra bởi các kích thích bên ngoài. Các bộ vi xử lý thường yêu cầu thêm các mạch phụ ( các IC giao tiếp nối tiếp, các IC điều khiển ngắt, các bộ định thời v.v... ) để thực hiện cùng các thao tác. Tuy nhiên, khả năng xử lý tuyệt đối của bộ vi điều khiển không bao giờ tiếp cận được với khả năng xử lý của bộ vi xử lý do một lượng lớn IC cần được cung cấp cho các chức năng trên *chip* – điều này trả giá cho khả năng xử lý.

Do kết cấu phần cứng bên trong các bộ vi điều khiển, các lệnh phải thật cô động và hầu hết được thực thi trên từng byte. Một tiêu chuẩn thiết kế là chương trình điều khiển cần phải đặt vừa trong ROM nội vì việc thêm ROM bên ngoài sẽ làm tăng giá thành của sản phẩm thiết kế trên bộ vi điều khiển. Một sơ đồ mã hóa chặt chẽ luôn luôn cần thiết cho tập lệnh. Đặc trưng này hiếm thấy ở các bộ vi xử lý; các kiểu định địa chỉ mạnh của chúng làm cho việc mã hóa các lệnh ít chặt chẽ hơn.

## 1.10 KHÁI NIỆM MỚI

Các bộ vi điều khiển không được dùng trong các máy tính nhưng lại được sử dụng trong các sản phẩm tiêu dùng và các sản phẩm công nghiệp. Những người sử dụng các sản phẩm này thường không nhận biết sự hiện diện của các bộ vi điều khiển; với họ, các thành phần bên trong là những chi tiết không quan trọng trong thiết kế.

Không giống như các hệ máy tính được xác định bởi khả năng được lập trình và được tái lập trình của chúng, các bộ vi điều khiển được lập trình thường trực cho một công việc. Sự so sánh này dẫn đến sự khác biệt hoàn toàn về cấu trúc giữa các hệ máy tính và các bộ vi điều khiển. Các hệ máy tính có tỉ lệ RAM-ROM rất cao sao cho các chương trình của người sử dụng được thực thi trong một không gian RAM tương đối lớn và các chương trình giao tiếp với phần cứng được thực thi trong một không gian ROM nhỏ. Các bộ vi điều khiển ngược lại có tỉ số ROM-RAM cao. Chương trình điều khiển tương đối lớn được lưu trong ROM trong khi RAM chỉ được sử dụng như một bộ nhớ tạm thời. Do chương trình điều khiển được lưu thường trực trong ROM, chương trình này còn được gọi là *firmware*. Dựa vào mức độ bền vững, chương trình điều khiển chứa trong ROM nằm ở khoảng giữa phần mềm – các chương trình trong RAM bị mất khi ta không cung cấp điện nữa – và phần cứng – các mạch vật lý. Sự khác nhau về phần mềm và phần cứng hơi giống với sự khác nhau giữa một trang giấy ( phần cứng ) với các chữ được viết lên trang giấy này ( phần mềm ). Ta có thể xem *firmware* như là một bức thư có dạng chuẩn, được thiết kế và được in ra cho một mục đích duy nhất.

## 1.11 ƯU VÀ KHUYẾT ĐIỂM : MỘT THÍ DỤ THIẾT KẾ

Các công việc được thực hiện bởi các bộ vi điều khiển thì không mới. Điều mới là các thiết kế được hiện thực với ít thành phần hơn so với các thiết kế trước đó. Các thiết kế trước đó yêu cầu vài chục hoặc thậm chí vài trăm IC để hiện thực nay chỉ có một ít thành phần trong đó bao gồm bộ vi điều khiển. Số thành phần được giảm bớt, hậu quả trực tiếp của tính khả lập trình của các bộ vi điều khiển và độ tích hợp cao trong công nghệ chế tạo vi mạch, thường chuyển thành thời gian phát triển ngắn hơn, giá thành khi sản xuất thấp hơn, công suất tiêu thụ thấp hơn và độ tin cậy cao hơn. Các hoạt động logic yêu cầu vài IC thường được hiện thực bên trong bộ vi điều khiển cùng với một chương trình điều khiển thêm vào.

Vấn đề ở đây là tốc độ. Các giải pháp dựa trên bộ vi điều khiển không bao giờ nhanh bằng giải pháp dựa trên các thành phần rời rạc. Những tình huống đòi hỏi phải đáp ứng thật nhanh đối với các sự kiện ( chiếm thiểu số trong các ứng dụng ) sẽ được quản lý tồi khi dựa vào các bộ vi điều khiển.

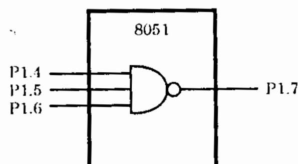
Lấy thí dụ ta khảo sát hình 1.6, thí dụ này hiện thực một cổng NAND bằng cách sử dụng *chip* vi điều khiển 8051. Không phải hoàn toàn hiển nhiên mà một bộ vi điều khiển có thể được sử dụng cho một hoạt động như vậy, nhưng lại có thể được. Phần mềm phải thực hiện các thao tác được chỉ ra trong lưu đồ ở hình 1.7. Chương trình hợp ngữ của 8051 cho hoạt động logic này như sau :

```
LOOP: MOV   C, P1.4     ; đọc bit P1.4 vào cờ nhớ
      ANL   C, P1.5     ; AND với P1.5
      ANL   C, P1.6     ; AND với P1.6
      CPL   C           ; đổi thành kết quả NAND
      MOV   P1.7, C     ; gởi kết quả vào P1.7
      SJMP  LOOP        ; lặp lại
```

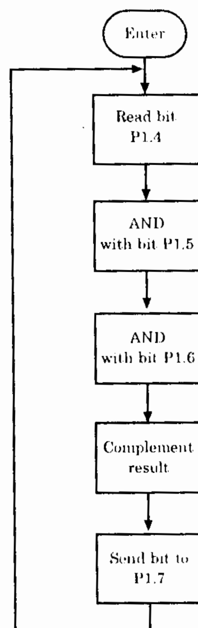
Nếu chương trình này được thực thi trên bộ vi điều khiển 8051, chức năng cổng NAND 3-ngõ vào được thực hiện. Thời gian trì hoãn truyền tính từ khi có sự chuyển trạng thái ở ngõ vào cho đến khi xuất hiện logic đúng ở ngõ ra khá lâu, ít nhất cũng so sánh được với các mạch TTL tương đương. Tùy thuộc vào lúc ngõ vào được chuyển trạng thái và chương trình biết được sự chuyển trạng thái này, thời gian trì hoãn có thể từ 3 đến 17 nsec ( giả sử 8051 hoạt động với tần số xung *clock* chuẩn là 12 MHz ). Thời gian trì hoãn truyền của mạch TTL tương đương vào

khoảng 10 nsec. Hiển nhiên là không có tranh cải khi so sánh tốc độ của các bộ vi điều khiển với các hiện thực TTL có cùng chức năng.

Trong nhiều ứng dụng, đặc biệt là các ứng dụng có liên quan đến con người, các khoảng thời gian trễ tính bằng nsec, µsec hoặc thậm chí msec là không quan trọng. Thí dụ về cổng logic ở trên minh họa rằng các bộ vi điều khiển có thể hiện thực các thao tác logic. Hơn nữa, khi các thiết kế trở nên phức tạp hơn, các lợi ích của thiết kế dựa trên các bộ vi điều khiển được thấy rõ hơn. Việc giảm bớt các thành phần là một điều lợi như đã đề cập, các thao tác trong chương trình điều khiển làm cho thiết kế có thể thay đổi bằng cách thay đổi phần mềm. Điều này có ảnh hưởng tối thiểu đến chu kỳ sản xuất.




Hình 1.6 : Hiện thực một cổng logic dùng bộ vi điều khiển 8051




Hình 1.7 : Lưu đồ cho chương trình hiện thực cổng logic

Enter : điểm nhập

Read bit P1.4 : đọc bit P1.4

AND with bit P1.5 : AND với bit P1.5

AND with bit P1.6 : AND với bit P1.6

Complement result : lấy bù kết quả

Send bit to P1.7 : gởi bit đến P1.7

# 2

# TÓM TẮT PHẦN CỨNG

## 2.1 TỔNG QUÁT

MCS-51 là họ vi điều khiển của Intel. Các nhà sản xuất IC khác như Siemens, Advanced Micro Devices, Fujitsu và Philips được cấp phép làm các nhà cung cấp thứ hai cho các *chip* của họ MCS-51.

Chương này giới thiệu về cấu trúc phần cứng của họ MCS-51. Tham khảo kỹ thuật của Intel cho các *chip* của họ MCS-51 được tìm thấy ở phụ lục E. Ta cần biết thêm nhiều chi tiết trong phụ lục này, thí dụ như các đặc tính điện chẳng hạn.

Nhiều đặc trưng phần cứng được minh họa bằng các chuỗi lệnh ngắn, các mô tả vắn tắt cũng được cung cấp cho từng thí dụ còn các chi tiết đầy đủ của tập lệnh được dành lại cho chương 3. Ta có thể tham khảo phụ lục A ( tóm tắt tập lệnh của 8051 ) và phụ lục C ( các định nghĩa cho từng lệnh của 8051 ).

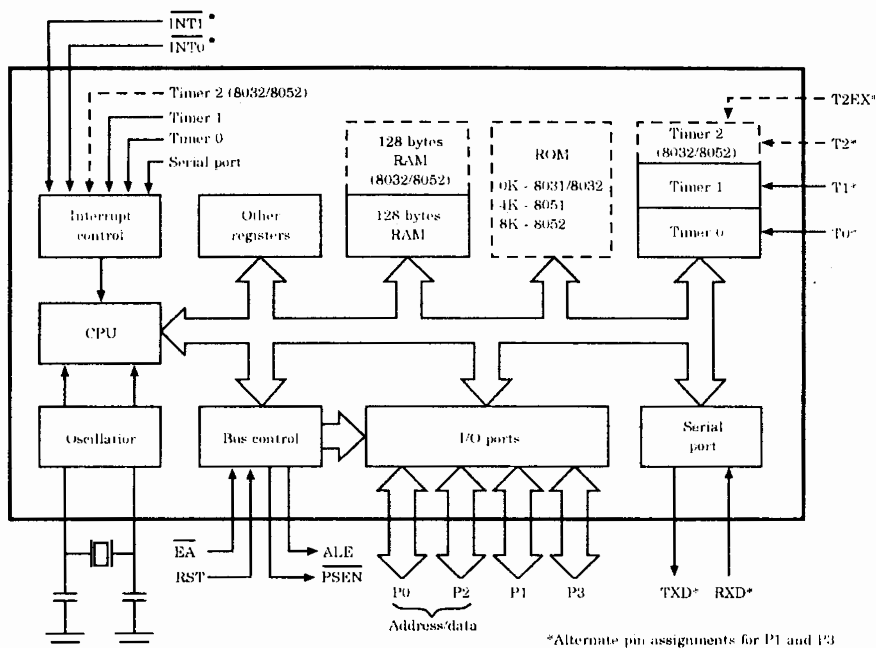
Vi mạch tổng quát của họ MCS-51 là *chip* 8051, linh kiện đầu tiên của họ này được đưa ra thị trường. *Chip* 8051 có các đặc trưng được tóm tắt như sau :

- 4KB ROM
- 128 byte RAM.
- 4 *port* xuất nhập ( I/O port ) 8-bit.
- 2 bộ định thời 16-bit.
- Mạch giao tiếp nối tiếp.
- Không gian nhớ chương trình ( mã ) ngoài 64K.
- Không gian nhớ dữ liệu ngoài 64K.
- Bộ xử lý bit ( thao tác trên các bit riêng rẽ )
- 210 vị trí nhớ được định địa chỉ, mỗi vị trí 1 bit.
- Nhân/chia trong 4μs.

Các thành viên khác của họ MCS-51 có các tổ hợp ROM ( EPROM ), RAM trên *chip* khác nhau hoặc có thêm bộ định thời thứ ba ( xem bảng 2.1 ). Mỗi một IC của họ MCS-51 cũng có phiên bản CMOS công suất thấp.

| ***Chip*** | **Bộ nhớ chương trình trên *chip*** | **Bộ nhớ dữ liệu trên *chip*** | **Các bộ định thời** |
|---|---|---|---|
| 8051 | 4 K ROM | 128 byte | 2 |
| 8031 | 0 K | 128 byte | 2 |
| 8751 | 4 K EPROM | 128 byte | 2 |
| 8052 | 8 K ROM | 256 byte | 3 |
| 8032 | 0 K | 256 byte | 3 |
| 8752 | 8 K EPROM | 256 byte | 3 |

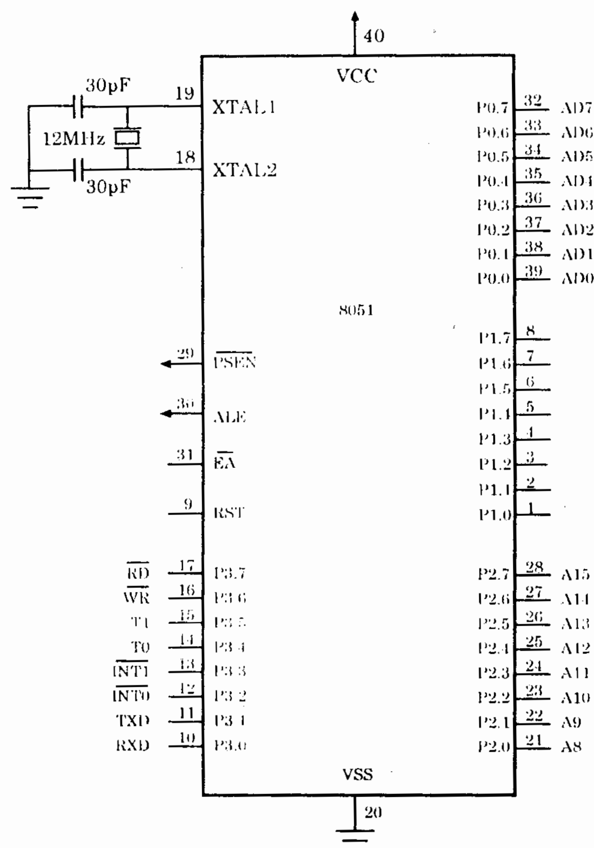
Bảng 2.1 : So sánh các *chip* của họ MCS-51




Hình 2.1 : Sơ đồ khối của *chip* 8051

Interrupt control : điều khiển ngắt
Other registers : các thanh ghi khác
128 bytes RAM : RAM 128 byte
Timer 2, 1, 0 : bộ định thời 2, 1, 0
CPU : đơn vị điều khiển trung tâm
Oscillator : mạch dao động
Bus control : điều khiển bus
I/O ports : các *port* xuất/nhập
Serial port : *port* nối tiếp
Address/data : địa chỉ/dữ liệu

Thuật ngữ " 8051 " được dùng để chỉ rộng rãi các *chip* của họ MCS-51. Khi việc thảo luận tập trung vào một cải tiến từ *chip* 8051 cơ bản, *chip* cải tiến được chỉ ra rõ ràng. Các đặc trưng vừa nêu trên được trình bày trong sơ đồ khối ở hình 2.1.




Hình 2.2 : Sơ đồ chân của 8051

## 2.2 CÁC CHÂN ( PINOUT )

Hình 2.2 cho ta sơ đồ chân của *chip* 8051. Mô tả tóm tắt chức năng của từng chân như sau.

Như ta thấy trong hình 2.2, 32 trong số 40 chân của 8051 có công dụng xuất/nhập, tuy nhiên 24 trong 32 đường này có 2 mục đích ( công dụng ) [ 26/32 đối với 8032/8052 ]. Mỗi một đường có thể hoạt động xuất/nhập hoặc hoạt động như một đường điều khiển hoặc hoạt động như một đường địa chỉ/dữ liệu của bus địa chỉ/dữ liệu đa hợp.

Các đặc trưng đã đề cập ở trên được trình bày trong sơ đồ khối ở hình 2.1.

32 chân nêu trên hình thành 4 *port* 8-bit. Với các thiết kế yêu cầu một mức tối thiểu bộ nhớ ngoài hoặc các thành phần bên ngoài khác, ta có thể sử dụng các *port* này làm nhiệm vụ xuất/nhập. 8 đường cho mỗi *port* có thể được xử lý như một đơn vị giao tiếp với các thiết bị song song như máy in, bộ biến đổi D-A, v.v... hoặc mỗi đường có thể hoạt động độc lập giao tiếp với một thiết bị đơn bit như chuyển mạch, LED, BJT, FET cuộn dây, động cơ, loa, v.v...

### 2.2.1 *Port* 0

*Port* 0 ( các chân từ 32 đến 39 trên 8051 ) có 2 công dụng. Trong các thiết kế có tối thiểu thành phần, *port* 0 được sử dụng làm nhiệm vụ xuất/nhập. Trong các thiết kế lớn hơn có bộ nhớ ngoài, *port* 0 trở thành bus địa chỉ và bus dữ liệu đa hợp [ byte thấp của bus địa chỉ nếu là địa chỉ ] ( xem 2.6 : Bộ nhớ ngoài ).

### 2.2.2 *Port* 1

*Port* 1 chỉ có một công dụng là xuất/nhập ( các chân từ 1 đến 8 trên 8051 ). Các chân của *port* 1 được ký hiệu là P1.0, P1.1, ... , P1.7 và được dùng để giao tiếp với thiết bị bên ngoài khi có yêu cầu. Không có chức năng nào khác nữa gán cho các chân của *port* 1, nghĩa là chúng chỉ được sử dụng để giao tiếp với các thiết bị ngoại vi.[ Ngoại lệ : với 8032/8052, ta có thể sử dụng P1.0 và P1.1 hoặc làm các đường xuất/nhập hoặc làm các ngõ vào cho mạch định thời thứ ba ].

### 2.2.3 *Port* 2

*Port* 2 ( các chân từ 21 đến 28 trên 8051 ) có 2 công dụng, hoặc làm nhiệm vụ xuất/nhập hoặc là byte địa chỉ cao của bus địa chỉ 16-bit cho các thiết kế có bộ nhớ chương trình ngoài hoặc các thiết kế có nhiều hơn 256 byte bộ nhớ dữ liệu ngoài.

### 2.2.4 *Port* 3

*Port* 3 ( các chân từ 10 đến 17 trên 8051 ) có 2 công dụng. Khi không hoạt động xuất/nhập, các chân của *port* 3 có nhiều chức năng riêng ( mỗi chân có chức năng riêng liên quan đến các đặc trưng cụ thể của 8051 ).

Bảng 2.2 dưới đây cho ta chức năng của các chân của *port* 3 và 2 chân P1.0, P1.1 của *port* 1.

| Bit | Tên | Địa chỉ bit | Chức năng |
|---|---|---|---|
| P3.0 | RxD | B0H | Chân nhận dữ liệu của *port* nối tiếp |
| P3.1 | TxD | B1H | Chân phát dữ liệu của *port* nối tiếp |
| P3.2 | $\overline{\text{INT0}}$ | B2H | Ngõ vào ngắt ngoài 0 |
| P3.3 | $\overline{\text{INT1}}$ | B3H | Ngõ vào ngắt ngoài 1 |
| P3.4 | T0 | B4H | Ngõ vào của bộ định thời/đếm 0 |
| P3.5 | T1 | B5H | Ngõ vào của bộ định thời/đếm 1 |
| P3.6 | $\overline{\text{WR}}$ | B6H | Điều khiển ghi bộ nhớ dữ liệu ngoài |
| P3.7 | $\overline{\text{RD}}$ | B7H | Điều khiển đọc bộ nhớ dữ liệu ngoài |
| P1.0 | T2 | 90H | Ngõ vào của bộ định thời/đếm 2 |
| P1.1 | T2EX | 91H | Nạp lại/thu nhận của bộ định thời 2 |

Bảng 2.2 : Chức năng của các chân của *port* 3 và *port* 1

### 2.2.5 Chân cho phép bộ nhớ chương trình $\overline{\text{PSEN}}$

8051 cung cấp cho ta 4 tín hiệu điều khiển bus. Tín hiệu cho phép bộ nhớ chương trình $\overline{\text{PSEN}}$ ( program store enable ) là tín hiệu xuất trên trên chân 29. Đây là tín hiệu điều khiển cho phép ta truy xuất bộ nhớ chương trình ngoài. Chân này thường nối với chân cho phép xuất $\overline{\text{OE}}$ ( output enable ) của EPROM ( hoặc ROM ) để cho phép đọc các byte lệnh.

Tín hiệu $\overline{\text{PSEN}}$ ở logic 0 trong suốt thời gian tìm-nạp lệnh. Các mã nhị phân của chương trình hay opcode ( mã thao tác ) được đọc từ EPROM, qua bus dữ liệu và được chốt vào thanh ghi lệnh IR của 8051 để được giải mã.

Khi thực thi một chương trình chứa ở ROM nội, $\overline{\text{PSEN}}$ được duy trì ở logic không tích cực ( logic 1 ).

### 2.2.6 Chân cho phép chốt địa chỉ ALE

8051 sử dụng chân 30, chân xuất tín hiệu cho phép chốt địa chỉ ALE ( address latch enable ) để giải đa hợp ( demultiplexing ) bus dữ liệu và bus địa chỉ. Khi *port* 0 được sử dụng làm bus địa chỉ/dữ liệu đa hợp, chân ALE xuất tín hiệu để chốt địa chỉ ( byte thấp của địa chỉ 16-bit ) vào một thanh ghi ngoài trong suốt ½ đầu của chu kỳ bộ nhớ ( memory cycle ). Sau khi điều này đã được thực hiện, các chân của *port* 0 sẽ xuất/nhập dữ liệu hợp hệ trong suốt ½ thứ hai của chu kỳ bộ nhớ.

Tín hiệu ALE có tần số bằng 1/6 tần số của mạch dao động bên trong *chip* vi điều khiển và có thể được dùng làm xung *clock* cho phần còn lại của hệ thống. Nếu mạch dao động có tần số 12 MHz, tín hiệu ALE có tần số 2 MHz. Ngoại hệ duy nhất là trong thời gian thực thi lệnh MOVX, một xung ALE sẽ bị bỏ qua ( xem hình 2.10 ). Chân ALE còn được dùng để nhận xung ngõ vào lập trình cho EPROM trên *chip* đối với các phiên bản của 8051 có EPROM này.

### 2.2.7 Chân truy xuất ngoài $\overline{EA}$

Ngõ vào này ( chân 31 ) có thể được nối với 5 V ( logic 1 ) hoặc với GND ( logic 0 ). Nếu chân này nối lên 5 V, 8051/8052 thực thi chương trình trong ROM nội ( chương trình nhỏ hơn 4K/8K ). Nếu chân này nối với GND ( và chân $\overline{PSEN}$ cũng ở logic 0 ), chương trình cần thực thi chứa ở bộ nhớ ngoài. Đối với 8031/8032 chân $\overline{EA}$ phải ở logic 0 vì chúng không có bộ nhớ chương trình trên *chip*. Nếu chân $\overline{EA}$ ở logic 0 đối với 8051/8052, ROM nội bên trong *chip* được vô hiệu hóa và chương trình cần thực thi chứa ở EPROM bên ngoài.

Các phiên bản EPROM của 8051 còn sử dụng chân $\overline{EA}$ làm chân nhận điện áp cấp điện 21V ( $V_{pp}$ ) cho việc lập trình EPROM nội ( nạp EPROM ).

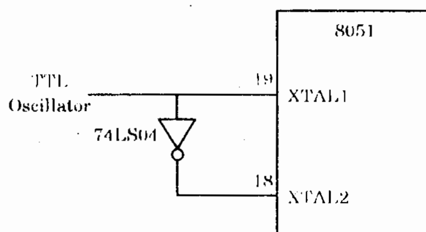
### 2.2.8 Chân RESET ( RST )

Ngõ vào RST ( chân 9 ) là ngõ vào xóa chính ( master reset ) của 8051 dùng để thiết lập lại trạng thái ban đầu cho hệ thống hay gọi tắt là *reset* hệ thống. Khi ngõ vào này được treo ở logic 1 tối thiểu hai chu kỳ máy, các thanh ghi bên trong của 8051 được nạp các giá trị thích hợp cho việc khởi động lại hệ thống ( xem mục 2.8 ).

### 2.2.9 Các chân XTAL1 và XTAL2

Như được vẽ trên hình 2.2, mạch dao động bên trong *chip* 8051 được ghép với thạch anh bên ngoài ở hai chân XTAL1 và XTAL2 ( chân 18 và

19 ). Các tụ ổn định cũng được yêu cầu như trên hình này. Tần số danh định của thạch anh là 12 MHz cho hầu hết các *chip* của họ MCS-51 ( 80C31BH-1 sử dụng thạch anh 16 MHz bên trong, mạch dao động trong *chip* không cần thạch anh bên ngoài ). Ở hình 2.3, 1 nguồn xung *clock* TTL có thể được nối với các chân XTAL1 và XTAL2.




Hình 2.3 8051 ghép với mạch dao động TTL bên ngoài

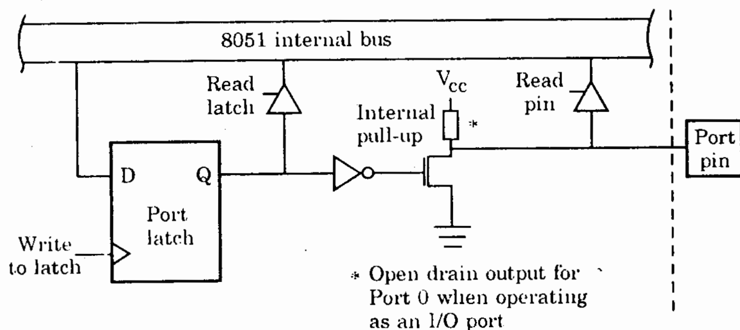
TTL oscillator : mạch dao động TTL

## 2.3 CẤU TRÚC CỦA *PORT* XUẤT/NHẬP

Sơ đồ mạch bên trong cho các chân của *port* xuất/nhập được vẽ đơn giản như ở hình 2.4. Việc ghi đến 1 chân của *port* sẽ nạp dữ liệu vào bộ chốt của *port*, ngõ ra Q của bộ chốt điều khiển một transistor trường và transistor này nối với chân của *port*. Khả năng *fanout* của các *port* 1, 2 và 3 là 4 tải vi mạch TTL loại Schottky công suất thấp ( LS ) còn của port 0 là 8 tải loại LS ( xem thêm chi tiết ở phụ lục E ).

Lưu ý là điện trở kéo lên ( pull up ) sẽ không có ở *port* 0 ( trừ khi *port* này làm nhiệm vụ của bus địa chỉ/dữ liệu đa hợp ), do vậy một điện trở kéo lên bên ngoài phải được cần đến.

Giá trị của điện trở này phụ thuộc vào đặc tính ngõ vào của thành phần ghép nối với chân của *port*.




Hình 2.4 : Mạch bên trong của các *port* xuất nhập

8051 internal bus : bus nội của 8051

Read latch : đọc bộ chốt

Internal pull up : kéo lên bên trong

Read pin : đọc chân *port*

*Port* pin : chân *port*

*Port* latch : bộ chốt của *port*

Write to latch : ghi vào bộ chốt

Ở đây ta thấy có cả 2 khả năng : "đọc bộ chốt" và "đọc chân *port*". Các lệnh yêu cầu thao tác đọc–sửa–ghi ( như lệnh CPL P1.5 ) đọc bộ chốt để tránh sự hiểu nhầm mức điện áp do sự kiện dòng tải tăng. Các lệnh nhập 1 bit của *port* ( như MOV C, P1.5 ) đọc chân *port*. Trong trường hợp này bộ chốt của *port* phải chứa 1 nếu không FET sẽ được kích bảo hòa và điều này kéo ngõ ra xuống mức thấp. Việc *reset* hệ thống sẽ *set* tất cả các bộ chốt *port*, do vậy các chân *port* có thể được dùng làm các ngõ nhập mà không cần phải *set* các bộ chốt *port* một cách tường minh. Tuy nhiên nếu một bộ chốt *port* bị xóa ( như CLR P1.5 ), chân *port* không thể làm nhiệm vụ tiếp theo là ngõ nhập trừ khi trước tiên ta phải *set* bộ chốt ( như SETB P1.5 ).

Hình 2.4 không trình bày mạch cho các chức năng khác của các *port* 0, 2 và 3. Khi các chức năng khác được sử dụng, các mạch kích ngõ ra được chuyển đến một địa chỉ nội ( *port* 2 ), địa chỉ/dữ liệu ( *port* 0 ) hoặc tín hiệu điều khiển ( *port* 3 ) tương ứng.

## 2.4 TỔ CHỨC BỘ NHỚ

Hầu hết các bộ vi xử lý ( CPU ) đều có không gian nhớ chung cho dữ liệu và chương trình. Điều này cũng hợp lý vì các chương trình thường được lưu trên đĩa và được nạp vào RAM để thực thi; vậy thì cả hai, dữ liệu và chương trình, đều lưu trú trong RAM.

Các *chip* vi điều khiển hiếm khi được sử dụng giống như các CPU trong các hệ máy tính, thay vào đó chúng được dùng làm thành phần trung tâm trong các thiết kế hướng điều khiển, trong đó bộ nhớ có dung lượng giới hạn, không có ổ đĩa và hệ điều hành. Chương trình điều khiển phải thường trú trong ROM.

Do lý do trên, 8051 có không gian bộ nhớ riêng cho chương trình và dữ liệu. Như ta đã thấy trong bảng 2.1 ở mục 2.1, cả 2 bộ nhớ chương trình và dữ liệu đều đặt bên trong *chip*, tuy nhiên ta có thể mở rộng bộ nhớ chương trình và bộ nhớ dữ liệu bằng cách sử dụng các *chip* nhớ bên ngoài với dung lượng tối đa là 64 K cho bộ nhớ chương trình ( hay bộ nhớ mã ) và 64 K cho bộ nhớ dữ liệu.

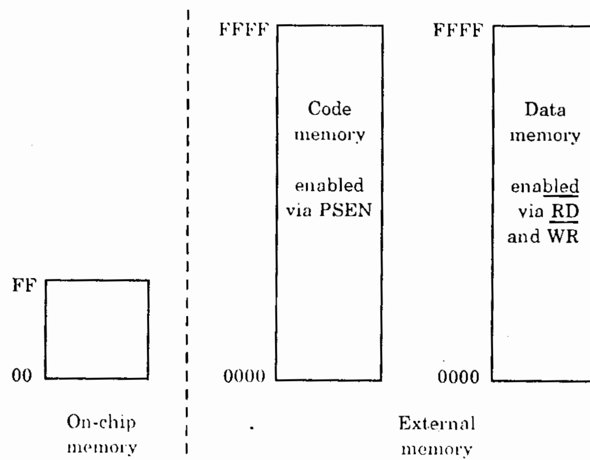
Bộ nhớ nội trong *chip* bao gồm ROM ( chỉ có ở 8051/8052 ) và RAM. RAM trên *chip* bao gồm vùng RAM đa chức năng ( nhiều công dụng ), vùng RAM với từng bit được định địa chỉ ( gọi tắt là vùng RAM định địa chỉ bit ), các dãy ( bank ) thanh ghi và các thanh ghi chức năng đặc biệt SFR ( special function register ). Hai đặc tính đáng lưu ý là :

(a) các thanh ghi và các *port* xuất/nhập được định địa chỉ theo kiểu ánh xạ bộ nhớ ( memory mapped ) và được truy xuất như một vị trí nhớ trong bộ nhớ.

(b) vùng *stack* thường trú trong RAM trên *chip* ( RAM nội ) thay vì ở trong RAM ngoài như đối với các bộ vi xử lý.

Hình 2.5 tóm tắt các không gian nhớ cho *chip* 8031 không có ROM nội, không trình bày chi tiết về bộ nhớ dữ liệu trên *chip* ( các cải tiến của 8032/8052 sẽ được tóm tắt sau ).

Hình 2.6 cho ta chi tiết của bộ nhớ dữ liệu trên *chip*. Ta thấy rằng không gian nhớ nội này được chia thành : các dãy thanh ghi ( 00H ÷ 1FH ), vùng RAM định địa chỉ bit ( 20H ÷ 2FH ), vùng RAM đa mục đích ( 30H ÷ 7FH ) và các thanh ghi chức năng đặc biệt ( 80H ÷ FFH ).




Hình 2.5 : Tóm tắt các không gian nhớ của *chip* 8031

On-chip memory : bộ nhớ trên *chip*

External memory : bộ nhớ ngoài

Code memory : bộ nhớ chương trình ( mã )

Enabled via $\overline{\text{PSEN}}$ : được cho phép bởi $\overline{\text{PSEN}}$

Data memory : bộ nhớ dữ liệu

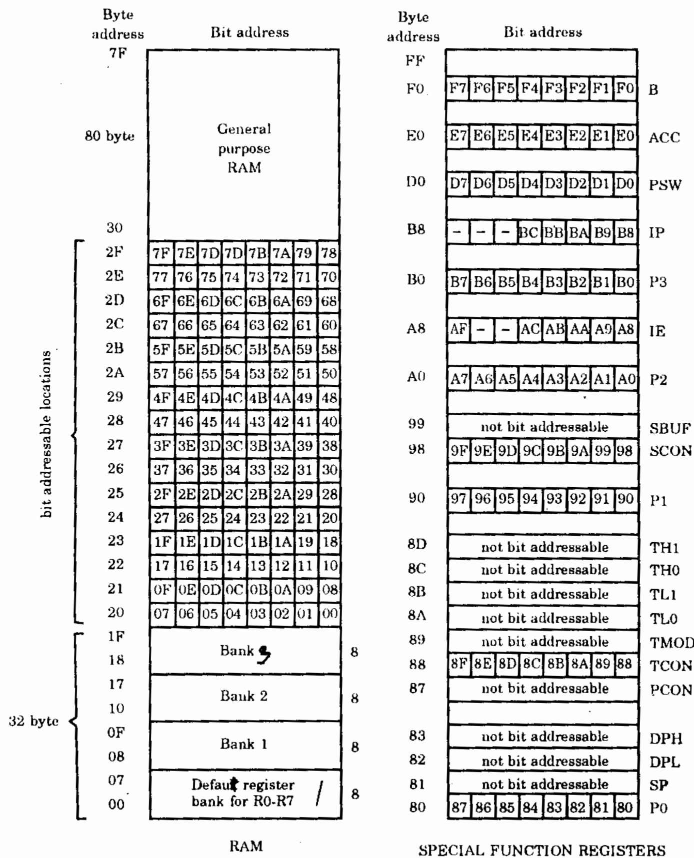
Enabled via $\overline{\text{RD}}$ and $\overline{\text{WR}}$ : được cho phép bởi $\overline{\text{RD}}$ và $\overline{\text{WR}}$

### 2.4.1 Vùng RAM đa mục đích

Mặc dù hình 2.6 trình bày vùng RAM đa mục đích có 80 byte đặt ở địa chỉ từ 30H đến 7FH, bên dưới vùng này từ địa chỉ 00H đến 2FH là vùng nhớ có thể được sử dụng tương tự ( mặc dù các vị trí nhớ này có các mục đích khác như sẽ thảo luận dưới đây ). Bất kỳ vị trí nhớ nào trong vùng RAM đa mục đích đều có thể được truy xuất tự do bằng cách sử dụng các kiểu định địa chỉ trực tiếp hoặc gián tiếp. Thí dụ để đọc nội dung tại địa chỉ 5FH của RAM nội vào thanh chứa A, ta dùng lệnh sau :

```
MOV  A, 5FH
```

Lệnh trên di chuyển 1 byte dữ liệu bằng cách dùng kiểu định địa chỉ trực tiếp để xác định vị trí nguồn ( nghĩa là địa chỉ 5FH ). Đích của dữ liệu được xác định rõ ràng trong opcode của lệnh là thanh chứa A ( các kiểu định địa chỉ sẽ được đề cập trong chương 3 ).




Hình 2.6 : Bộ nhớ dữ liệu trên *chip* 8051

Byte address, bit address : địa chỉ byte, địa chỉ bit

General purpose RAM : vùng RAM đa mục đích

Bank : dãy

Default register bank for R0 – R7 : dãy thanh ghi mặc định R0 – R7

Special function registers : các thanh ghi chức năng đặc biệt

Not bit addressable : không định địa chỉ bit

Vùng RAM đa mục đích còn có thể được truy xuất bằng cách dùng kiểu định địa chỉ gián tiếp qua các thanh ghi R0, R1. Thí dụ hai lệnh sau thực hiện cùng công việc như lệnh ở thí dụ trên :

```
MOV  R0, #5FH
MOV  A, @R0
```

Lệnh đầu tiên sử dụng kiểu định địa chỉ tức thời di chuyển giá trị 5FH vào thanh ghi R0, lệnh tiếp theo sử dụng kiểu định địa chỉ gián tiếp di chuyển dữ liệu trỏ bởi R0 vào thanh chứa A.

### 2.4.2 Vùng RAM định địa chỉ bit

8051 chứa 210 vị trí bit được định địa chỉ trong đó 128 bit chứa trong các byte ở địa chỉ từ 20H đến 2FH [ 16 byte x 8 bit = 128 bit ] và phần còn lại chứa trong các thanh ghi chức năng đặc biệt.

Ý tưởng truy xuất các bit riêng rẽ thông qua phần mềm là một đặc trưng mạnh của hầu hết các bộ vi điều khiển. Các bit có thể được *set*, xóa, AND, OR, v.v... bằng một lệnh. Hầu hết các bộ vi xử lý yêu cầu một chuổi lệnh đọc–sửa–ghi để nhận được cùng một kết quả. Ngoài ra 8051 còn có các *port* xuất/nhập có thể định địa chỉ từng bit, điều này làm đơn giản việc giao tiếp bằng phần mềm với các thiết bị xuất/nhập đơn bit.

Như vừa đề cập ở trên, 8051 có 128 vị trí bit được định địa chỉ và có nhiều mục đích ở các byte có địa chỉ từ 20H đến 2FH. Các địa chỉ này được truy xuất như là các byte hay các bit tùy vào lệnh cụ thể. Thí dụ để *set* bit 67H bằng 1 ta dùng lệnh sau :

```
SETB  67H
```

Tham chiếu hình 2.6 ta thấy bit ở địa chỉ 67H là bit có ý nghĩa lớn nhất của byte ở địa chỉ 2CH. Lệnh vừa nêu trên không ảnh hưởng đến các bit khác trong byte này. Hầu hết các bộ vi xử lý muốn thực hiện công việc như trên phải dùng các lệnh có dạng tương tự như sau :

```
MOV  A, 2CH            ; đọc cả byte
ORL  A, #10000000B     ; set bit có ý nghĩa lớn nhất
MOV  2CH, A            ; ghi trở lại cả byte
```

### 2.4.3 Các dãy thanh ghi

32 vị trí thấp nhất của bộ nhớ nội chứa các dãy thanh ghi. Các lệnh của 8051 hỗ trợ 8 thanh ghi từ R0 đến R7 thuộc dãy 0 ( bank 0 ). Đây là dãy mặc định sau khi *reset* hệ thống. Các thanh ghi này ở các địa chỉ từ 00H đến 07H. Lệnh sau đây đọc nội dung tại địa chỉ 05H vào thanh chứa :

MOV A, R5

Lệnh này là lệnh 1-byte dùng kiểu định địa chỉ thanh ghi. Dĩ nhiên thao tác tương tự có thể được thực hiện với 1 lệnh 2-byte bằng cách dùng kiểu định địa chỉ trực tiếp :

MOV A, 05H

Các lệnh sử dụng các thanh ghi từ R0 đến R7 là các lệnh ngắn và thực hiện nhanh hơn so với các lệnh tương đương sử dụng kiểu định địa chỉ trực tiếp. Các giá trị dữ liệu thường được sử dụng nên chứa ở một trong các thanh ghi này. Dãy thanh ghi đang được sử dụng được gọi là dãy thanh ghi tích cực. Dãy thanh ghi tích cực có thể được thay đổi bằng cách thay đổi các bit chọn dãy trong từ trạng thái chương trình PSW ( sẽ đề cập sau ). Giả sử rằng dãy thanh ghi 3 ( bank 3 ) tích cực, lệnh sau đây ghi nội dung của thanh chứa A vào vị trí 18H :

MOV R0, A

Ý tưởng "các dãy thanh ghi" cho phép "chuyển đổi ngữ cảnh" nhanh và có hiệu quả ở những nơi mà các phần riêng rẽ của phần mềm sử dụng một tập thanh ghi riêng, độc lập với các phần khác của phần mềm.

## 2.5 CÁC THANH GHI CHỨC NĂNG ĐẶC BIỆT ( SFR )

Các thanh ghi nội của hầu hết các bộ vi xử lý đều được truy xuất rõ ràng bởi một tập lệnh. Thí dụ lệnh INCA của *chip* 6809 tăng nội dung của thanh chứa A bởi 1. Thao tác được xác định rõ ràng trong opcode của lệnh. Việc truy xuất các thanh ghi cũng được sử dụng trên 8051. Lệnh INC A thực hiện cùng công việc trên.

Các thanh ghi nội của 8051 được cấu hình thành một phần của RAM trên *chip*, do vậy mỗi một thanh ghi cũng có một địa chỉ. Điều này hợp lý với 8051 vì *chip* này có rất nhiều thanh ghi. Cũng như các thanh ghi từ R0 đến R7, ta có 21 thanh ghi chức năng đặc biệt SFR chiếm phần trên của RAM nội từ địa chỉ 80H đến FFH ( xem hình 2.6 ).

Lưu ý là không phải tất cả 128 địa chỉ từ 80h đến FFH đều được định nghĩa mà chỉ có 21 địa chỉ được định nghĩa [ 26 trên 8032/8052 ].

Thanh chứa A có thể được truy xuất rõ ràng như được minh họa trong các thí dụ ở các phần trên. Hầu hết các thanh ghi chức năng đặc biệt được truy xuất bằng kiểu định địa chỉ trực tiếp. Trong hình 2.6 ta cần lưu ý là một số thanh ghi chức năng đặc biệt được định địa chỉ từng bit và định địa chỉ từng byte. Thí dụ ta có lệnh sau :

```
SETB 0E0H
```

sẽ *set* bit 0 của thanh chứa A lên 1, các bit khác của thanh chứa không thay đổi. Ta nhận thấy rằng tại địa chỉ E0H có thể là : địa chỉ byte cho cả thanh chứa và địa chỉ bit của bit có ý nghĩa thấp nhất trong thanh chứa. Vì lệnh SETB thao tác trên các bit và không thao tác trên các byte, chỉ có bit được định địa chỉ bị ảnh hưởng. Lưu ý là các bit được định địa chỉ trong một thanh ghi chức năng đặc biệt có 5 bit cao của địa chỉ giống nhau cho tất cả các bit của thanh ghi này. Lấy thí dụ *port* 1 có địa chỉ byte là 90H ( hay 10010000B ) và các bit trong *port* này có các địa chỉ từ 90H tới 97H hay 10010xxxB.

Từ trạng thái chương trình PSW ( program status word ) sẽ được thảo luận chi tiết trong mục sau. Các thanh ghi chức năng đặc biệt khác cũng được giới thiệu vắn tắt, các chi tiết về chúng sẽ được đề cập trong các chương tiếp theo.

### 2.5.1 Từ trạng thái chương trình PSW

| Bit | Ký hiệu | Địa chỉ | Mô tả bit |
|---|---|---|---|
| PSW.7 | CY | D7H | Cờ nhớ |
| PSW.6 | AC | D6H | Cờ nhớ phụ |
| PSW.5 | F0 | D5H | Cờ 0 |
| PSW.4 | RS1 | D4H | Chọn dãy thanh ghi ( bit 1 ) |
| PSW.3 | RS0 | D3H | Chọn dãy thanh ghi ( bit 0 ) |
| | | | 00 = bank 0 : địa chỉ từ 00H đến 07H |
| | | | 01 = bank 1 : địa chỉ từ 08H đến 0FH |
| | | | 10 = bank 2 : địa chỉ từ 10H đến 17H |
| | | | 11 = bank 3 : địa chỉ từ 18H đến 1FH |
| PSW.2 | OV | D2H | Cờ tràn |
| PSW.1 | – | D1H | Dự trữ |
| PSW.0 | P | D0H | Cờ kiểm tra chẵn lẻ |

Bảng 2.3 : Thanh ghi PSW

PSW có địa chỉ là D0H chứa các bit trạng thái có chức năng được tóm tắt trong bảng 2.3. Từng bit của PSW được khảo sát dưới đây :

**Cờ nhớ**

Cờ nhớ CY ( carry flag ) có 2 công dụng. Công dụng truyền thống trong các phép toán số học là được *set* bằng 1 nếu có số nhớ từ phép cộng bit 7 hoặc có số mượn mang đến bit 7. Thí dụ nếu thanh chứa A có nội dung là FFH, lệnh :

```
ADD  A, #1
```

sẽ làm cho thanh chứa A có nội dung là 00H và cờ CY trong PSW được *set* bằng 1. Cờ nhớ CY còn là thanh chứa logic được dùng như một thanh ghi 1-bit đối với các lệnh logic thao tác trên các bit. Lấy thí dụ lệnh sau đây sẽ AND bit 25H với cờ nhớ CY và đặt kết quả trở về cờ nhớ :

```
ANL  C, 25H     ; AND bit ở địa chỉ 25H với cờ nhớ
```

**Cờ nhớ phụ**

Khi cộng các giá trị BCD, cờ nhớ phụ AC ( auxiliary carry flag ) được *set* bằng 1 nếu có một số nhớ được tạo ra từ bit 3 chuyển sang bit 4 hoặc nếu kết quả trong đề-cát thấp nằm trong tầm từ 0AH đến 0FH. Nếu các giá trị được cộng là giá trị BCD, lệnh cộng phải được tiếp theo bởi lệnh DA A ( hiệu chỉnh thập phân thanh chứa A ) để đưa các kết quả lớn hơn 9 về giá trị đúng.

**Cờ 0**

Đây là cờ có nhiều mục đích dành cho các ứng dụng của người lập trình.

**Các bit chọn dãy thanh ghi**

Các bit chọn dãy thanh ghi RS0, RS1 dùng để xác định dãy thanh ghi tích cực. Các bit này được xóa sau khi có thao tác *reset* hệ thống và đổi mức logic bởi phần mềm khi cần. Thí dụ ba lệnh sau cho phép dãy thanh ghi 3 ( bank 3 ) tích cực, sau đó di chuyển nội dung của R7 ( địa chỉ byte 1FH ) vào thanh chứa A :

```
SETB  RS1
SETB  RS0
MOV   A, R7
```

Khi đoạn chương trình trên được dịch, các địa chỉ bit sẽ thay thế cho các ký hiệu RS0 và RS1, vậy thì lệnh SETB RS1 tương đương với lệnh SETB 0D4H.

**Cờ tràn**

Cờ tràn OV ( overflow flag ) được *set* bằng 1 sau phép toán cộng hoặc trừ nếu có xuất hiện một tràn số học. Khi các số có dấu được cộng hoặc được trừ, phần mềm có thể kiểm tra bit tràn OV để xác định xem kết quả có nằm trong tầm hay không.

Với phép cộng các số không dấu, cờ tràn OV được bỏ qua. Kết quả lớn hơn +128 hoặc nhỏ hơn −127 sẽ *set* cờ OV bằng 1. Thí dụ phép cộng sau đây gây ra 1 tràn và *set* cờ OV trong PSW :

| Số hex : | | Số thập phân : | |
|---|---|---|---|
| | 0F | | 15 |
| + | 7F | | +127 |
| | 8E | | 142 |

8EH biểu diễn số âm −116, như vậy không đúng với kết quả mong muốn là 142 nên cờ OV được *set* bằng 1.

**Cờ chẵn lẻ**

Bit chẵn lẻ P tự động được *set* bằng 1 hay xóa bằng 0 ở mỗi chu kỳ máy để thiết lập kiểm tra chẵn cho thanh chứa A. Số các bit 1 trong thanh chứa cộng với bit P luôn luôn là số chẵn. Thí dụ nếu thanh chứa có nội dung 10101101B, bit P sẽ là 1 để có số bit 1 là 6. Bit chẵn lẻ được sử dụng nhiều để kết hợp với các chương trình xuất/nhập nối tiếp trước khi truyền dữ liệu hoặc để kiểm tra chẵn lẻ sau khi nhận dữ liệu.

### 2.5.2 Thanh ghi B

Thanh ghi B ở địa chỉ F0H được dùng chung với thanh chứa A trong các phép toán nhân, chia. Lệnh MUL AB nhân 2 số 8-bit không dấu chứa trong A và B và chứa kết quả 16-bit vào cặp thanh ghi B:A ( thanh chứa A cất byte thấp và thanh ghi B cất byte cao ).

Lệnh chia DIV AB chia A bởi B, thương số cất trong thanh chứa A và dư số cất trong thanh ghi B. Thanh ghi B còn được xử lý như 1 thanh ghi nháp. Các bit được định địa chỉ của thanh ghi B có địa chỉ từ F0H đến F7H.

### 2.5.3 Con trỏ *stack*

Con trỏ *stack* SP ( stack pointer ) là 1 thanh ghi 8-bit ở địa chỉ 81H. SP chứa địa chỉ của dữ liệu hiện đang ở đỉnh của *stack*. Các lệnh liên quan đến *stack* bao gồm lệnh cất dữ liệu vào *stack* và lệnh lấy dữ liệu ra khỏi *stack*. Việc cất vào *stack* làm tăng SP trước khi ghi dữ liệu và việc lấy dữ liệu ra khỏi *stack* sẽ giảm SP. Vùng *stack* của 8051 được giữ trong RAM nội và được giới hạn đến các địa chỉ truy xuất được bởi kiểu định

địa chỉ gián tiếp. Vùng RAM nội có 128 byte trên 8031/8051 hoặc 256 byte trên 8032/8052; nếu ta khởi động SP để bắt đầu vùng *stack* ở địa chỉ 60H bằng lệnh :

```
MOV  SP, #5FH
```

vùng *stack* được giới hạn là 32 byte trên 8031/8051 vì địa chỉ cao nhất của RAM trên *chip* là 7FH. Giá trị 5FH được dùng ở đây vì SP tăng lên 60H trước khi thao tác cất vào *stack* đầu tiên được thực thi.

Nếu ta không khởi động SP, nội dung mặc định của thanh ghi này là 07H nhằm duy trì sự tương thích với 8048, bộ vi điều khiển tiền nhiệm của 8051. Kết quả là thao tác cất vào *stack* đầu tiên sẽ lưu dữ liệu vào vị trí nhớ có địa chỉ 08H. Như vậy nếu phần mềm ứng dụng không khởi động SP, dãy thanh ghi 1 ( và có lẽ 2 và 3 ) không còn hợp lệ vì vùng này được sử dụng làm *stack*. Các lệnh PUSH và POP sẽ cất dữ liệu vào *stack* và lấy dữ liệu từ *stack*, các lệnh gọi chương trình con ( ACALL, LCALL ) và lệnh trở về ( RET, RETI ) cũng cất và phục hồi nội dung của bộ đếm chương trình PC ( program counter ).

### 2.5.4 Con trỏ dữ liệu DPTR

Con trỏ dữ liệu DPTR ( data pointer ) được dùng để truy xuất bộ nhớ chương trình ngoài hoặc bộ nhớ dữ liệu ngoài. DPTR là 1 thanh ghi 16-bit có địa chỉ là 82H ( DPL, byte thấp ) và 83H ( DPH, byte cao ). Ba lệnh sau đây ghi 55H vào RAM ngoài ở địa chỉ 1000H :

```
MOV  A, #55H
MOV  DPTR, #1000H
MOV  @DPTR, A
```

Lệnh đầu tiên sử dụng kiểu định địa chỉ tức thời để nạp hằng dữ liệu 55H vào thanh chứa A. Lệnh thứ hai cũng sử dụng kiểu định địa chỉ tức thời, lần này nạp hằng địa chỉ 16-bit 1000H cho con trỏ dữ liệu DPTR. Lệnh thứ ba sử dụng kiểu định địa chỉ gián tiếp di chuyển giá trị 55H chứa trong thanh chứa A đến RAM ngoài tại địa chỉ chứa trong DPTR ( 1000H ).

### 2.5.5 Các thanh ghi *port*

Các *port* xuất nhập của 8051 bao gồm *port* 0 tại địa chỉ 80H, *port* 1 tại địa chỉ 90H, *port* 2 tại địa chỉ A0H và *port* 3 tại địa chỉ B0H. Các *port* 0, 2 và 3 không được dùng để xuất/nhập nếu ta sử dụng thêm bộ nhớ ngoài hoặc nếu có một số đặc tính đặc biệt của 8051 được sử dụng ( như là ngắt, *port* nối tiếp, ... ). P1.2 đến P1.7, ngược lại, luôn luôn là các đường xuất/nhập đa mục đích hợp lệ.

Tất cả *port* đều được định địa chỉ từng bit nhằm cung cấp các khả năng giao tiếp mạnh. Thí dụ ta có một động cơ nối qua một cuộn dây và một mạch kích dùng transistor nối tới bit 7 của *port* 1, động cơ có thể ngưng hay chạy chỉ nhờ vào một lệnh đơn của 8051 :

```
SETB P1.7
```

làm động cơ chạy và

```
CLR P1.7
```

làm động cơ ngưng.

Các lệnh trên sử dụng toán tử **.** ( dot ) để định địa chỉ 1 bit trong 1 byte, cho phép định địa chỉ từng bit.

Trình dịch hợp ngữ thực hiện biến đổi dạng ký hiệu thành địa chỉ thực tế, nghĩa là hai lệnh sau tương đương :

```
CLR P1.7
CLR 97H
```

Việc sử dụng các ký hiệu được định nghĩa trước ( tiền định nghĩa ) của trình dịch hợp ngữ sẽ được thảo luận chi tiết trong các tài liệu về lập trình hợp ngữ trên họ MCS-51 hoặc ở chương 7.

Thí dụ sau đây khảo sát việc giao tiếp với 1 thiết bị có bit trạng thái gọi là BUSY, bit này được *set* bằng 1 khi thiết bị đang bận và được xóa khi thiết bị đã sẵn sàng. Nếu BUSY được nối với bit 5 của *Port* 1, vòng lặp sau đây được dùng để chờ cho đến khi thiết bị sẵn sàng :

```
WAIT: JB P1.5, WAIT
```

Lệnh trên có nghĩa là nếu bit P1.5 được *set*, nhảy đến nhãn WAIT ( cũng có nghĩa là nhảy về và kiểm tra lần nữa ).

### 2.5.6 Các thanh ghi định thời

8051 có 2 bộ đếm/định thời ( timer/counter ) 16-bit để định các khoảng thời gian hoặc để đếm các sự kiện. Bộ định thời 0 có địa chỉ 8AH ( TL0, byte thấp ) và 8CH ( TH0, byte cao ); bộ định thời 1 có địa chỉ 8BH ( TL1, byte thấp ) và 8DH ( TH1, byte cao ).

Hoạt động của bộ định thời được thiết lập bởi thanh ghi chế độ định thời TMOD ( timer mode register ) ở địa chỉ 89H và thanh ghi điều khiển định thời TCON ( timer control register ) ở địa chỉ 88H. Chỉ có TCON được định địa chỉ từng bit.

Các bộ định thời sẽ được thảo luận chi tiết sau.

### 2.5.7 Các thanh ghi của *port* nối tiếp

Bên trong 8051 có một *port* nối tiếp để truyền thông với các thiết bị nối tiếp như các thiết bị đầu cuối hoặc *modem*, hoặc để giao tiếp với các IC khác có mạch giao tiếp nối tiếp ( như các thanh ghi dịch chẳng hạn ). Một thanh ghi được gọi là bộ đệm dữ liệu nối tiếp SBUF ( serial data buffer ) ở địa chỉ 99H lưu giữ dữ liệu truyền đi và dữ liệu nhận về. Việc ghi lên SBUF sẽ nạp dữ liệu để truyền và việc đọc SBUF sẽ lấy dữ liệu đã nhận được.

Các chế độ hoạt động khác nhau được lập trình thông qua thanh ghi điều khiển *port* nối tiếp SCON ( serial port control register ) ở địa chỉ 98H, thanh ghi này được định địa chỉ từng bit.

Hoạt động chi tiết của *port* nối tiếp sẽ mô tả sau.

### 2.5.8 Các thanh ghi ngắt

8051 có một cấu trúc ngắt với 2 mức ưu tiên và 5 nguyên nhân ngắt. ( 5 source, 2 priority level interrupt structure ). Các ngắt bị vô hiệu hóa sau khi *reset* hệ thống và sau đó được cho phép bằng cách ghi vào thanh ghi cho phép ngắt IE ( interrupt enable register ) ở địa chỉ A8H. Mức ưu tiên ngắt được thiết lập qua thanh ghi ưu tiên ngắt IP ( interrupt priority register ) ở địa chỉ B8H. Cả 2 thanh ghi này đều được định địa chỉ từng bit.

Các ngắt sẽ được đề cập chi tiết sau.

### 2.5.9 Thanh ghi điều khiển nguồn

Thanh ghi điều khiển nguồn PCON ( power control register ) có địa chỉ 87H chứa các bit điều khiển được tóm tắt trong bảng 2.4.

Bit SMOD tăng gấp đôi tốc độ baud của *port* nối tiếp khi *port* này hoạt động ở các chế độ 1, 2 hoặc 3. Các bit 4, 5 và 6 của PCON không được định nghĩa. Các bit 2 và 3 là các bit cờ đa mục đích dành cho các ứng dụng của người sử dụng.

Các bit điều khiển nguồn, nguồn giảm PD và nghỉ IDL, hợp lệ trong tất cả *chip* thuộc họ MCS-51, nhưng chỉ được hiện thực trong các phiên bản CMOS của MCS-51. PCON không được định địa chỉ bit.

#### Chế độ nguồn giảm

Lệnh thiết lập bit PD bằng 1 sẽ là lệnh sau cùng được thực thi trước khi đi vào chế độ nguồn giảm. Ở chế độ nguồn giảm :

(1) mạch dao động trên *chip* ngừng hoạt động

(2) mọi chức năng ngừng hoạt động

(3) nội dung của RAM trên *chip* được duy trì

(4) các chân *port* duy trì mức logic của chúng

(5) ALE và $\overline{PSEN}$ được giữ ở mức thấp. Chỉ ra khỏi chế độ này bằng cách *reset* hệ thống.

Trong suốt thời gian ở chế độ nguồn giảm, Vcc có điện áp là 2V. Cần phải giữ cho Vcc không thấp hơn sau khi đạt được chế độ nguồn giảm và cần phục hồi Vcc = 5V tối thiểu 10 chu kỳ dao động trước khi chân RST đạt mức thấp lần nữa.

| **Bit** | **Ký hiệu** | **Mô tả** |
|---|---|---|
| 7 | SMOD | Bit tăng gấp đôi tốc độ baud, bit này khi *set* làm cho tốc độ baud tăng 2 ở các chế độ 1, 2 và 3 của *port* nối tiếp |
| 6 | – | Không định nghĩa |
| 5 | – | Không định nghĩa |
| 4 | – | Không định nghĩa |
| 3 | GF1 | Bit cờ đa mục đích 1 |
| 2 | GF0 | Bit cờ đa mục đích 2 |
| 1 | PD | Nguồn giảm; thiết lập để tích cực chế độ nguồn giảm, chỉ ra khỏi chế độ bằng *reset*. |
| 0 | IDL | Chế độ nghỉ; thiết lập để tích cực chế độ nghỉ, chỉ ra khỏi chế độ bằng 1 ngắt hoặc *reset* hệ thống. |

Bảng 2.4 : Thanh ghi PCON

### Chế độ nghỉ

Lệnh thiết lập bit IDL bằng 1 sẽ là lệnh sau cùng được thực thi trước khi đi vào chế độ nghỉ. Ở chế độ nghỉ, tín hiệu *clock* nội được khóa không cho đến CPU nhưng không khóa đối với các chức năng ngắt, định thời và *port* nối tiếp. Trạng thái của CPU được duy trì và nội dung của tất cả các thanh ghi cũng được giữ không đổi.

Các chân *port* cũng được duy trì các mức logic của chúng. ALE và $\overline{PSEN}$ được giữ ở mức cao.

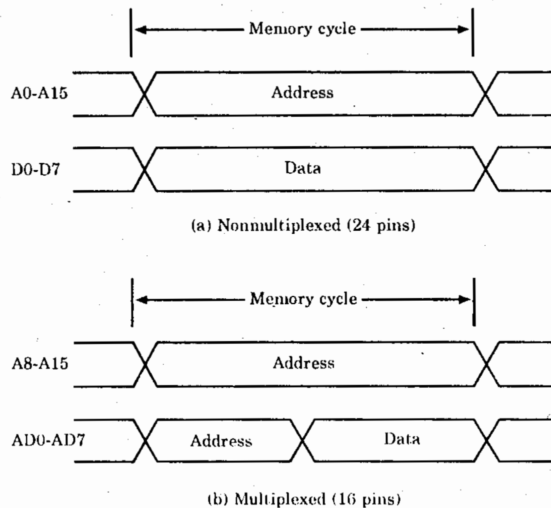
Chế độ nghỉ kết thúc bằng cách cho phép ngắt hoặc bằng cách *reset* hệ thống. Cả hai cách vừa nêu đều xóa bit IDL.

## 2.6 BỘ NHỚ NGOÀI

Các bộ vi điều khiển cần có khả năng mở rộng các tài nguyên trên *chip* ( bộ nhớ, I/O, v.v... để tránh hiện tượng cổ chai trong thiết kế. Cấu trúc của MCS-51 cho ta khả năng mở rộng không gian bộ nhớ chương trình đến 64K và không gian bộ nhớ dữ liệu đến 64K. ROM và RAM ngoài được thêm vào khi cần.

Các IC giao tiếp ngoại vi cũng có thể được thêm vào để mở rộng khả năng xuất/nhập. Chúng trở thành 1 phần của không gian bộ nhớ dữ liệu ngoài bằng cách sử dụng cách định địa chỉ kiểu I/O ánh xạ bộ nhớ. Khi bộ nhớ ngoài được sử dụng, *port* 0 không làm nhiệm vụ của *port* xuất/nhập, *port* này trở thành bus địa chỉ ( A0 – A7 ) và bus dữ liệu ( D0 – D7 ) đa hợp. Ngõ ra ALE chốt byte thấp của địa chỉ ở thời điểm bắt đầu mỗi một chu kỳ bộ nhớ ngoài. *Port* 2 thường ( nhưng không phải luôn luôn ) được dùng làm byte cao của bus địa chỉ.

Trước khi thảo luận các chi tiết cụ thể về các bus địa chỉ và dữ liệu đa hợp, ý tưởng tổng quát được trình bày ở hình 2.7.




Hình 2.7 : Đa hợp bus địa chỉ ( byte thấp ) và bus dữ liệu (a) không đa hợp ( 24 chân ) (b) đa hợp ( 16 chân )

Memory cycle : chu kỳ bộ nhớ

Address : địa chỉ

Data : dữ liệu

Nonmultiplexed : không đa hợp

Multiplexed : đa hợp

Sắp xếp không đa hợp sử dụng 16 đường địa chỉ và 8 đường dữ liệu, tổng cộng 24 đường. Sắp xếp đa hợp kết hợp 8 đường của bus dữ liệu và byte thấp của bus địa chỉ, do vậy ta chỉ cần 16 đường.

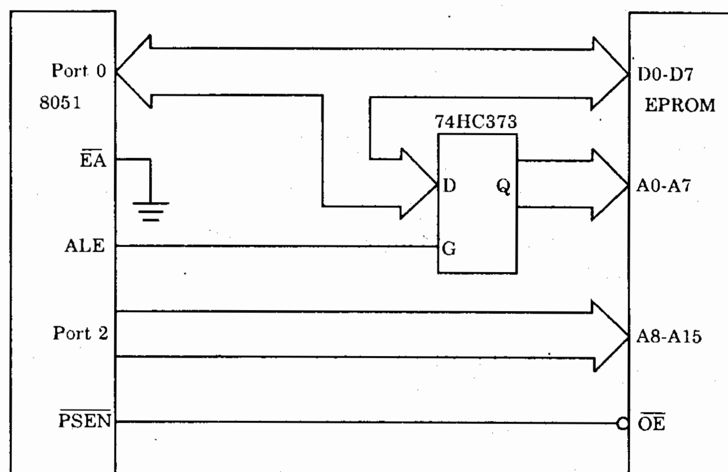
Việc tiết kiệm các chân cho phép ta đóng gói bộ vi điều khiển họ MCS-51 trong 1 vỏ 40 chân.

Sắp xếp đa hợp có hoạt động như sau : trong ½ chu kỳ đầu của chu kỳ bộ nhớ, byte thấp của địa chỉ được cung cấp bởi *port* 0 và được chốt nhờ tín hiệu ALE. Mạch chốt 74HC373 giữ cho byte thấp của địa chỉ ổn định trong cả chu kỳ bộ nhớ. Trong ½ sau của chu kỳ bộ nhớ, *port* 0 được sử dụng làm bus dữ liệu và dữ liệu được đọc hay ghi.

### 2.6.1 Truy xuất bộ nhớ chương trình ngoài

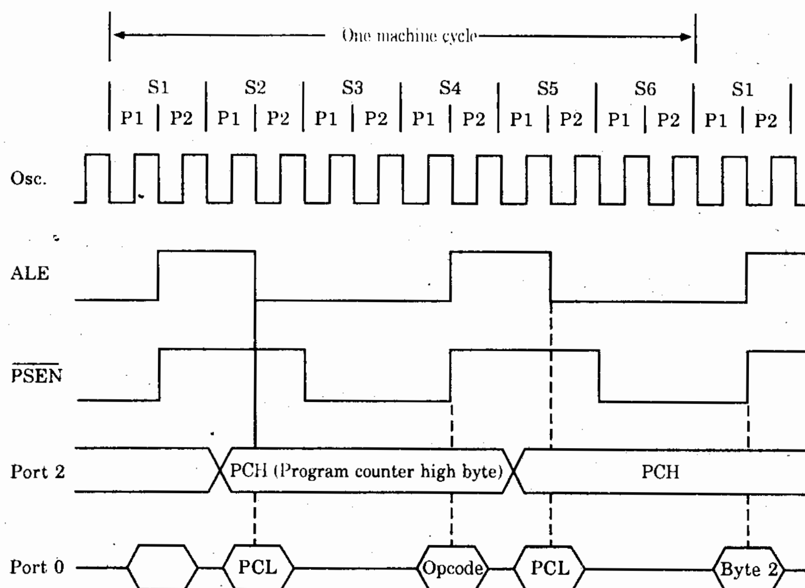
Bộ nhớ chương trình ngoài là bộ nhớ chỉ đọc, được cho phép bởi tín hiệu $\overline{PSEN}$. Khi có 1 EPROM ngoài được sử dụng, cả hai *port* 0 và *port* 2 đều không còn là các *port* xuất/nhập.

Kết nối phần cứng với bộ nhớ ngoài EPROM được trình bày ở hình 2.8.




Hình 2.8 : Truy xuất bộ nhớ chương trình ngoài

Một chu kỳ máy của 8051 có 12 chu kỳ dao động. Nếu bộ dao động trên *chip* có tần số 12 MHz, một chu kỳ máy dài 1 μs. Trong 1 chu kỳ máy điển hình, ALE có 2 xung và 2 byte của lệnh được đọc từ bộ nhớ chương trình ( nếu lệnh chỉ có 1 byte, byte thứ hai được loại bỏ ). Giản đồ thời gian của chu kỳ máy này, được gọi là chu kỳ tìm-nạp lệnh được trình bày ở hình 2.9.




Hình 2.9 : Giản đồ thời gian của chu kỳ tìm-nạp lệnh ở bộ nhớ ngoài

One machine cycle : một chu kỳ máy

Program counter high byte : byte cao của PC

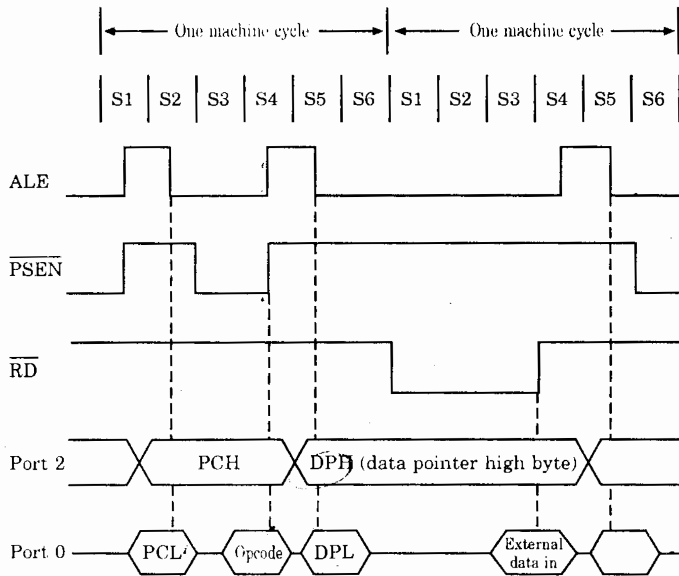
## 2.6.2 Truy xuất bộ nhớ dữ liệu ngoài

Bộ nhớ dữ liệu ngoài là bộ nhớ đọc/ghi được cho phép bởi các tín hiệu $\overline{RD}$ và $\overline{WR}$ ở các chân P3.7 và P3.6. Lệnh dùng để truy xuất bộ nhớ dữ liệu ngoài là MOVX, sử dụng hoặc con trỏ dữ liệu 16-bit DPTR hoặc R0, R1 làm thanh ghi chứa địa chỉ.

RAM có thể giao tiếp với 8051 theo cùng cách như EPROM ngoại trừ đường $\overline{RD}$ nối với đường cho phép xuất ( $\overline{OE}$ ) của RAM và $\overline{WR}$ nối với đường ghi ( $\overline{W}$ ) của RAM. Các kết nối với bus dữ liệu và bus địa chỉ giống như EPROM. Bằng cách sử dụng các *port* 0 và *port* 2 như ở phần trên, ta có 1 dung lượng RAM ngoài lên đến 64 K được kết nối với 8051.

Giản đồ thời gian của thao tác đọc dữ liệu ở bộ nhớ dữ liệu ngoài được trình bày ở hình 2.10 cho lệnh MOVX A, @DPTR. Lưu ý là cả 2 xung ALE và $\overline{PSEN}$ được bỏ qua ở nơi mà xung $\overline{RD}$ cho phép đọc RAM [ nếu lệnh MOVX và RAM ngoài không bao giờ được dùng, các xung ALE luôn có tần số bằng 1/6 tần số của mạch dao động ].

Giản đồ thời gian của chu kỳ ghi ( lệnh MOVX @DPTR, A ) cũng tương tự ngoại trừ các xung $\overline{WR}$ ở mức thấp và dữ liệu được xuất ra ở *port* 0 ( $\overline{RD}$ vẫn ở mức cao ).



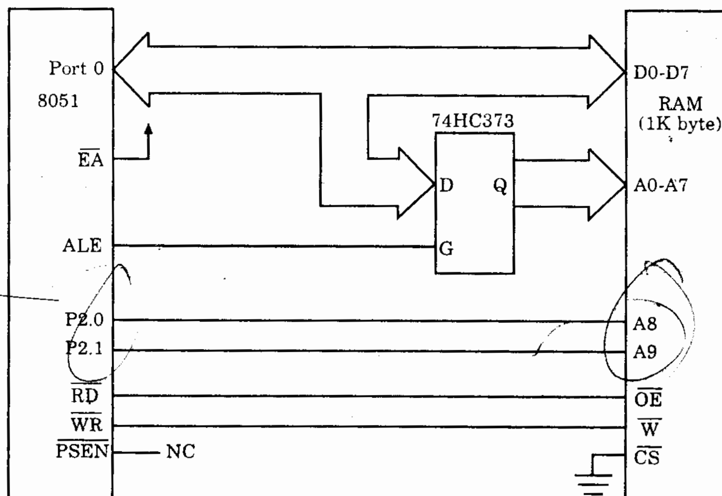

Hình 2.10 : Giản đồ thời gian của lệnh MOVX

One machine cycle : một chu kỳ máy

Data pointer high byte : byte cao của DPTR

External data in : nhập dữ liệu từ bộ nhớ ngoài

Opcode : mã thao tác




Hình 2.11 : Giao tiếp với 1 K RAM

*Port* 2 giảm bớt được chức năng làm nhiệm vụ cung cấp byte cao của địa chỉ trong các hệ thống tối thiểu thành phần, hệ thống không dùng bộ nhớ chương trình ngoài và chỉ có 1 dung lượng nhỏ bộ nhớ dữ liệu ngoài. Các địa chỉ 8 bit có thể truy xuất bộ nhớ dữ liệu ngoài với cấu hình bộ nhớ nhỏ hướng trang ( page-oriented ). Nếu có nhiều hơn 1 trang 256-byte RAM, 1 vài bit từ *port* 2 ( hoặc 1 *port* khác ) có thể chọn 1 trang. Thí dụ với 1 RAM 1KB ( nghĩa là 4 trang 256 byte ), ta có thể kết nối RAM này với 8051 như ở hình 2.11.

Các bit 0 và 1 của *port* 2 phải được khởi động để chọn 1 trang, rồi lệnh MOVX được dùng để đọc hoặc ghi trên trang này. Thí dụ ta giả sử P2.0 = P2.1 = 0, các lệnh sau có thể dùng để đọc các nội dung của RAM ngoài ở địa chỉ 0050H vào thanh chứa A :

```
MOV   R0, #50H
MOVX  A, @R0
```

Để đọc ở địa chỉ cuối cùng của RAM này, 03FFH, trang 3 được chọn nghĩa là ta phải *set* cho các bit P2.0 và P2.1 bằng 1. Chuỗi lệnh sau được dùng :

```
SETB  P2.0
SETB  P2.1
MOV   R0, #0FFH
MOVX  A, @R0
```

Một đặc trưng của thiết kế này là các bit từ 2 đến 7 của *port* 2 không còn cần làm bit địa chỉ nữa, các bit còn lại này có thể sử dụng cho mục đích xuất/nhập.

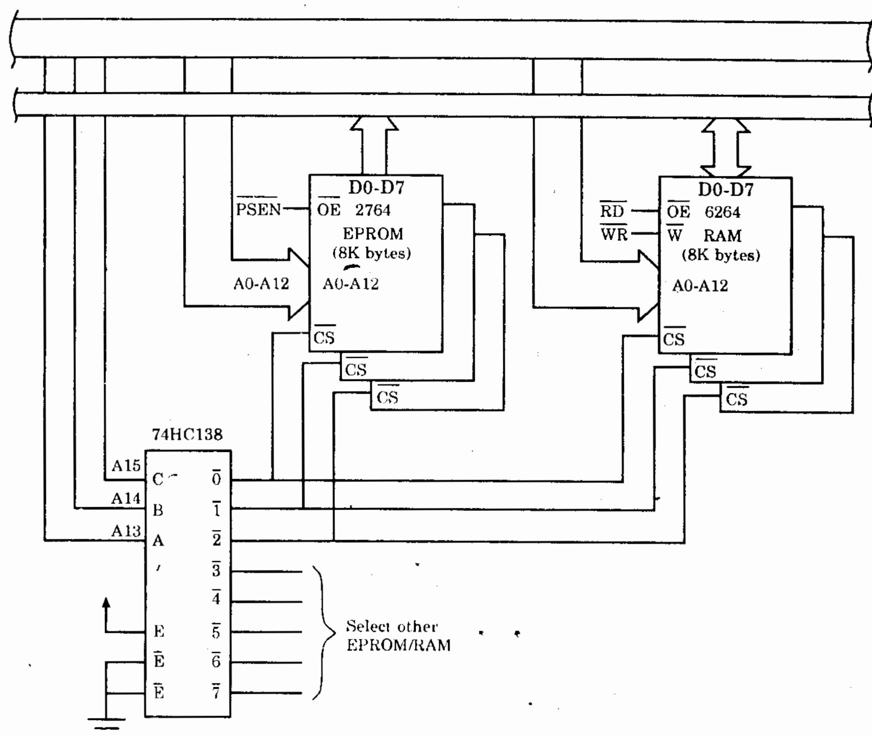
### 2.6.3 Giải mã địa chỉ

Nếu có nhiều EPROM hoặc nhiều RAM hoặc cả 2 giao tiếp với 8051 ta cần phải giải mã địa chỉ. Việc giải mã này cũng cần cho hầu hết các bộ vi xử lý.

Thí dụ nếu các RAM và ROM 8 KB được sử dụng, địa chỉ phải được giải mã để chọn các IC nhớ này trên các giới hạn 8 K : 0000H – 1FFFH, 2000H – 3FFFH, ...

Một IC giải mã điển hình là 74HC138 được dùng với các ngõ ra được nối với các ngõ vào chọn *chip* $\overline{CS}$ của các IC nhớ như được mô tả ở hình 2.12 cho một bộ nhớ có nhiều EPROM 2764 ( 8K ) và RAM 6264 ( 8K ). Cần lưu ý là do các đường cho phép riêng rẽ ( $\overline{PSEN}$ cho bộ nhớ

chương trình, $\overline{RD}$ và $\overline{WR}$ cho bộ nhớ dữ liệu ), 8051 có thể quản lý không gian nhớ đến 64K cho bộ nhớ EPROM và 64K cho bộ nhớ RAM.




Hình 2.12 : Giải mã địa chỉ

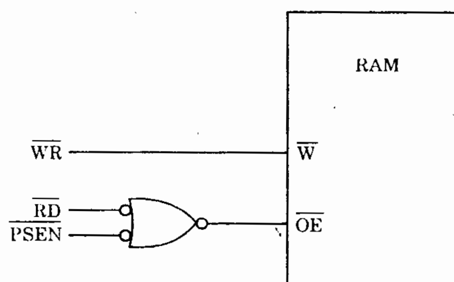
### 2.6.4 Các không gian nhớ chương trình và dữ liệu gối nhau

Vì bộ nhớ chương trình là bộ nhớ chỉ đọc, một tình huống khó xử được phát sinh trong quá trình phát triển phần mềm cho 8051. Làm thế nào phần mềm được viết cho một hệ thống đích để gỡ rối nếu phần mềm chỉ có thể được thực thi từ không gian bộ nhớ chương trình chỉ đọc.

Giải pháp tổng quát là cho các không gian bộ nhớ chương trình và dữ liệu ngoài gối lên nhau. Vì $\overline{PSEN}$ được dùng để đọc bộ nhớ chương trình và $\overline{RD}$ được dùng để đọc bộ nhớ dữ liệu, một RAM có thể chiếm không gian nhớ chương trình và dữ liệu bằng cách nối chân $\overline{OE}$ tới ngõ ra cổng AND có các ngõ vào là $\overline{PSEN}$ và $\overline{RD}$.

Mạch trình bày ở hình 2.13 cho phép IC RAM được ghi như là bộ nhớ dữ liệu và được đọc như là bộ nhớ chương trình hoặc dữ liệu. Vậy thì một chương trình có thể được nạp vào RAM ( bằng cách ghi vào

RAM như là bộ nhớ dữ liệu ) và được thực thi (bằng cách truy xuất như bộ nhớ chương trình ).




Hình 2.13 : Gối 2 không gian nhớ chương trình và dữ liệu

## 2.7 CÁC CẢI TIẾN CỦA 8032 / 8052

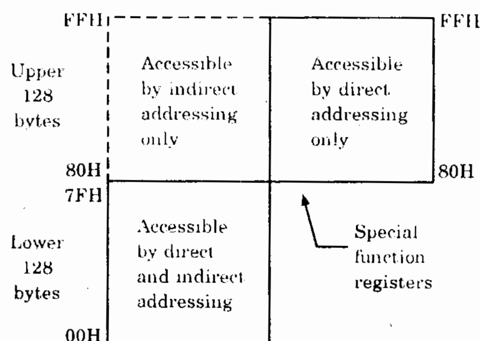
Các vi mạch 8032 / 8052 ( và các phiên bản CMOS ) có hai cải tiến so với 8031/8051. Một là có thêm 128 byte RAM trên *chip* từ địa chỉ 80H đến FFH. Điều này không xung đột với các thanh ghi chức năng đặc biệt ( có cùng địa chỉ ) vì 128 byte RAM thêm vào chỉ có thể truy xuất bằng cách dùng kiểu định địa chỉ gián tiếp. Một lệnh như sau :

```
MOV  A, 0F0H
```

di chuyển nội dung của thanh ghi B tới thanh chứa A trên các IC của họ MCS-51. Chuỗi lệnh :

```
MOV  R0, #0F0H
MOV  A, @R0
```

đọc vào thanh ghi A nội dung tại địa chỉ F0H trên các IC 8032/8052 nhưng không được định nghĩa trên 8031/8051. Tổ chức bộ nhớ nội của 8032/8052 được tóm tắt ở hình 2.14.




Hình 2.14 : Không gian nhớ nội của 8032/8052

Upper 128 bytes : 128 byte trên

Lower 128 bytes : 128 byte dưới

Accessible by indirect addressing only : chỉ được truy xuất bằng kiểu định địa chỉ gián tiếp

Accessible by direct addressing only : chỉ được truy xuất bằng kiểu định địa chỉ trực tiếp

Accessible by direct and indirect addressing : truy xuất bằng kiểu định địa chỉ trực tiếp và gián tiếp

| **Thanh ghi** | **Địa chỉ** | **Mô tả** | **Địa chỉ bit** |
|---|---|---|---|
| T2CON | C8H | Điều khiển | Có |
| RCAP2L | CAH | Nhận byte thấp | Không |
| RCAP2H | CBH | Nhận byte cao | Không |
| TL2 | CCH | Byte thấp của bộ định thời 2 | Không |
| TH2 | CDH | Byte cao của bộ định thời 2 | Không |

Bảng 2.5 : Các thanh ghi của bộ định thời 2

Cải tiến thứ 2 là có thêm bộ định thời 16-bit. Bộ định thời 2 này được lập trình nhờ vào 5 thanh ghi chức năng đặc biệt thêm vào. Chúng được tóm tắt trong bảng 2.5 và sẽ được mô tả chi tiết sau trong chương 4 ( hoạt động định thời ).

## 2.8 HOẠT ĐỘNG *RESET*

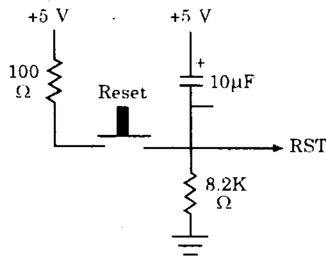
8051 được *reset* bằng cách giữ chân RST ở mức cao tối thiểu 2 chu kỳ máy và sau đó chuyển về mức thấp. RST có thể được tác động bằng tay hoặc được tác động khi cấp nguồn bằng cách dùng một mạch RC như được trình bày ở hình 2.15. Trạng thái của tất cả các thanh ghi sau khi *reset* hệ thống được tóm tắt trong bảng 2.6.

Quan trọng nhất trong các thanh ghi này có lẽ là thanh ghi PC ( bộ đếm chương trình ), được nạp 0000H. Khi RST trở lại mức thấp, việc thực thi chương trình luôn luôn bắt đầu ở vị trí đầu tiên trong bộ nhớ chương trình: địa chỉ 0000H. Nội dung của RAM trên *chip* không bị ảnh hưởng bởi hoạt động *reset*.

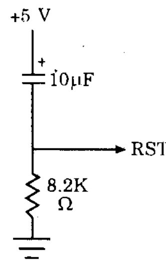
| **Thanh ghi** | **Nội dung** |
|---|---|
| Bộ đếm chương trình | 0000H |
| Thanh chứa A | 00H |
| Thanh ghi B | 00H |
| PSW | 00H |

| | |
|---|---|
| SP | 07H |
| DPTR | 0000H |
| *Port* 0 – 3 | FFH |
| IP | xxx00000B ( 8031/8051 ) |
| | xx000000B ( 8032/8052 ) |
| IE | 0xx00000B ( 8031/8051 ) |
| | 0x000000B ( 8032/8052 ) |
| Các thanh ghi định thời | 00H |
| SCON | 00H |
| SBUF | 00H |
| PCON ( HMOS ) | 0xxxxxxxB |
| PCON ( CMOS ) | 0xxx0000B |

Bảng 2.6 : Giá trị của các thanh ghi sau khi *reset* hệ thống




(a) Manual reset




(b) Power-on reset

Hình 2.15 : Hai mạch dùng *reset* hệ thống (a) *reset* bằng tay (b) *reset* khi cấp nguồn

Reset : nút nhất *reset*

Manual reset : *reset* bằng tay

Power-on reset : *reset* khi cấp nguồn

# 3

# TÓM TẮT TẬP LỆNH

## 3.1 MỞ ĐẦU

Chương này giới thiệu tập lệnh ( instruction set ) của họ MCS-51 thông qua việc khảo sát các kiểu định địa chỉ ( addressing mode ) và các thí dụ dựa trên các tình huống lập trình điển hình. Phụ lục A cho ta bảng tóm tắt tất cả các lệnh của 8051. Phụ lục C mô tả chi tiết từng lệnh. Các phụ lục được dùng để tham khảo khi đọc chương này. Chương này chưa bàn đến kỹ thuật lập trình cũng như hoạt động của trình dịch hợp ngữ ( assembler ), chương trình được dùng để chuyển đổi chương trình nguồn viết bằng hợp ngữ thành chương trình ngôn ngữ máy. Các chủ đề này được thảo luận sau trong các chương 7 và 8.

Tập lệnh của MCS-51 được tối ưu hóa cho các ứng dụng điều khiển 8-bit. Nhiều kiểu định địa chỉ cô đọng và nhanh dùng để truy xuất RAM nội được dùng đến nhằm tạo thuận lợi cho các thao tác trên các cấu trúc dữ liệu nhỏ. Tập lệnh cũng hỗ trợ các biến 1-bit cho phép quản lý bit trực tiếp trong các hệ logic và điều khiển có yêu cầu xử lý bit.

Cũng như các bộ vi xử lý 8-bit, các lệnh của 8051 có các opcode 8-bit, do vậy số lệnh có thể lên đến 256 lệnh ( thực tế có 255 lệnh, 1 lệnh không được định nghĩa ) . Ngoài opcode, một số lệnh còn có thêm 1 hoặc 2 byte nữa cho dữ liệu hoặc địa chỉ. Tập lệnh có 139 lệnh 1 byte, 92 lệnh 2-byte và 24 lệnh 3-byte. Phụ lục B trình bày opcode của các lệnh. Phụ lục này cho ta thấy, với từng lệnh một : opcode, mã gợi nhớ, số byte của lệnh và số chu kỳ máy cần để thực thi lệnh.

## 3.2 CÁC KIỂU ĐỊNH ĐỊA CHỈ

Khi một lệnh được thực thi và lệnh này cần dữ liệu, một câu hỏi được đặt ra là : " Dữ liệu chứa ở đâu ". Câu trả lời cho câu hỏi này tạo ra các kiểu định địa chỉ của 8051. Có nhiều kiểu định địa chỉ do vậy có nhiều câu trả lời cho câu hỏi nêu trên, chẳng hạn như : trong byte thứ 2

*của 1 lệnh, trong thanh ghi R4, trong địa chỉ trực tiếp hoặc có thể trong* bộ nhớ ngoài ở địa chỉ chứa trong con trỏ dữ liệu.

Các kiểu định địa chỉ là phần cần thiết cho toàn bộ tập lệnh của mỗi một bộ vi xử lý, bộ vi điều khiển. Các kiểu định địa chỉ cho phép ta xác định rõ nguồn và đích của dữ liệu theo nhiều cách khác nhau phụ thuộc vào tình huống lập trình. Trong phần này chúng ta sẽ khảo sát tất cả các kiểu định địa chỉ của 8051 và nêu thí dụ cho từng kiểu. Có 8 kiểu định địa chỉ :

- Thanh ghi ( register ).
- Trực tiếp ( direct ).
- Gián tiếp ( indirect ).
- Tức thời ( immediate ).
- Tương đối ( relative ).
- Tuyệt đối ( absolute ).
- Dài ( long ).
- Chỉ số ( indexed ).

### 3.2.1 Định địa chỉ thanh ghi

Người lập trình trên 8051 có thể truy xuất 8 thanh ghi " làm việc " được đánh số từ R0 đến R7. Các lệnh sử dụng kiểu định địa chỉ thanh ghi được mã hóa bằng cách dùng 3 bit thấp nhất của opcode ( của lệnh ) để chỉ ra một thanh ghi bên trong không gian địa chỉ logic này. Vậy thì một mã chức năng và địa chỉ toán hạng có thể kết hợp để hình thành một lệnh ngắn ( 1-byte ) [ xem hình 3.1.a ].

Hợp ngữ của 8051 chỉ ra kiểu định địa chỉ thanh ghi bằng ký hiệu Rn, trong đó n có giá trị từ 0 đến 7. Thí dụ để cộng nội dung của thanh ghi R7 với thanh chứa A, ta dùng lệnh sau :

```
ADD  A, R7
```

và lệnh này có opcode là 00101111B. Năm bit cao 00101 cho biết đây là lệnh cộng và 3 bit thấp 111 chỉ ra thanh ghi R7. Ta có thể tham khảo phụ lục C để xác nhận điều vừa đề cập.

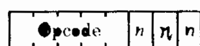
Có 4 dãy thanh ghi " làm việc " nhưng ở 1 thời điểm chỉ có 1 dãy tích cực. Các dãy thanh ghi chiếm 32 byte đầu tiên của RAM dữ liệu trên *chip* ( địa chỉ từ 00H đến 1FH ) và ta dùng các bit 4 và 3 của từ trạng thái chương trình PSW để chỉ ra dãy thanh ghi tích cực. Một *reset*

(Có thể dg Using

bằng phần cứng cho phép dãy 0 tích cực còn các dãy khác được chọn bằng cách sửa đổi các bit 4 và 3 của PSW sao cho phù hợp. Thí dụ lệnh :

**MOV PSW, #00011000B**

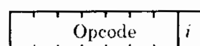
sẽ tích cực dãy thanh ghi 3 bằng cách *set* các bit chọn dãy thanh ghi ( RS1 và RS0 ) trong PSW lên 1 ( các bit này ở vị trí 4 và 3 ).




(a) Register addressing (e.g., ADD A, R5)




(b) Direct addressing (e.g., ADD A, direct)




(c) Indirect addressing (e.g., ADD A, @R0)



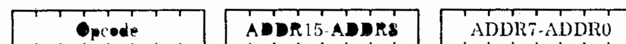

(d) Immediate addressing (e.g., ADD A, #55H)



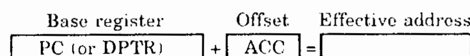

(e) Relative addressing (e.g., SJMP <dest>)




(f) Absolute addressing (e.g., AJMP <dest>)




(g) Long addressing (e.g., LJMP <dest>)




(h) Indexed addressing (e.g., MOVC A,@A + PC)

Hình 3.1 : Các kiểu định địa chỉ của 8051 (a) định địa chỉ thanh ghi (b) định địa chỉ trực tiếp (c) định địa chỉ gián tiếp (d) định địa chỉ tức thời (e) định địa chỉ tương đối (f) định địa chỉ tuyệt đối (g) định địa chỉ dài (h) định địa chỉ chỉ số

Register addressing : định địa chỉ thanh ghi ( thí dụ : ADD A, R5 )

Direct addressing : định địa chỉ trực tiếp ( thí dụ : ADD A, direct )

Indirect addressing : định địa chỉ gián tiếp ( thí dụ : ADD A, @R0 )

Immediate addressing : định địa chỉ tức thời ( thí dụ : ADD A, #55H )

Relative addressing : định địa chỉ tương đối ( thí dụ : SJMP <dest> )

Absolute addressing : định địa chỉ tuyệt đối ( thí dụ : AJMP <dest> )

Long addressing : định địa chỉ dài ( thí dụ : LJMP <dest> )

Indexed addressing : định địa chỉ chỉ số ( thí dụ : MOVC A, @A+PC )

Một số lệnh đặc biệt liên quan đến 1 thanh ghi xác định nào đó như là thanh chứa A, con trỏ dữ liệu, ... không cần các bit địa chỉ, bản thân opcode của lệnh đã chỉ ra thanh ghi cần thiết. Các lệnh đặc biệt liên quan đến thanh ghi này tham chiếu đến thanh chứa bằng ký hiệu " A ", con trỏ dữ liệu bằng ký hiệu " DPTR ", bộ đếm chương trình bằng ký hiệu " PC ", cờ nhớ bằng ký hiệu " C " và cặp thanh ghi AB bằng ký hiệu " AB ". Lấy thí dụ :

```
INC  DPTR
```

là lệnh 1-byte, lệnh này cộng 1 vào nội dung của con trỏ dữ liệu 16-bit. Opcode của lệnh này được cho ở phụ lục C.

### 3.2.2 Định địa chỉ trực tiếp

Kiểu định địa chỉ trực tiếp được sử dụng để truy xuất các biến nhớ hoặc các thanh ghi trên *chip*. Một byte thêm vào tiếp theo opcode dùng để xác định địa chỉ ( xem hình 3.1.b ).

Phụ thuộc vào bit có giá trị vị trí ( hay trọng số ) cao của địa chỉ trực tiếp, một trong hai không gian nhớ trên *chip* được chọn. Khi bit 7 bằng 0, địa chỉ trực tiếp ở trong tầm từ 0 đến 127 ( 00H – 7FH ) và 128 byte thấp của RAM trên *chip* được tham chiếu. Tất cả các *port* xuất/nhập và các thanh ghi chức năng đặc biệt, điều khiển, trạng thái được gán địa chỉ trong tầm từ 128 đến 255 ( 80H – FFH ). Khi byte địa chỉ theo sau opcode có nội dung nằm trong giới hạn này ( với bit 7 bằng 1 ), thanh ghi chức năng đặc biệt được truy xuất. Thí dụ *port* 0 và *port* 1 được gán địa chỉ trực tiếp là 80H và 90H.

Ta không nhất thiết phải biết địa chỉ của các thanh ghi này, trình dịch hợp ngữ cho phép ta sử dụng mã gợi nhớ viết tắt dễ hiểu như là " P0 " cho *port* 0, " TMOD " cho thanh ghi chế độ định thời ( timer mode register ) v.v... Lệnh sau đây là một thí dụ cho kiểu định địa chỉ trực tiếp :

```
MOV  P1, A
```

lệnh này chuyển nội dung của thanh chứa A vào *port* 1. Địa chỉ trực tiếp của *port* 1 là 90H được xác định bởi trình dịch hợp ngữ và trình dịch này đặt 90H vào byte 2 của lệnh. Nguồn của dữ liệu, thanh chứa, được xác định rõ ràng trong opcode.

Bằng cách sử dụng phụ lục C, ta thấy lệnh vừa nêu trên có mã là :

| | | |
|---|---|---|
| 10001001 | – | byte 1 : opcode |
| 10010000 | – | byte 2 : địa chỉ của P1 ( 90H ) |

### 3.2.3 Định địa chỉ gián tiếp

Làm cách nào nhận biết 1 biến khi địa chỉ của biến đã được xác định, được tính toán hoặc được sửa đổi trong khi một chương trình đang chạy ? Tình huống này phát sinh khi ta quản lý các vị trí nhớ liên tiếp, các điểm nhập được định chỉ số trong các bảng chứa trong RAM, các số chính xác hoặc các chuỗi ký tự. Các kiểu định địa chỉ thanh ghi hoặc trực tiếp không sử dụng được cho các tình huống này, do vậy ta cần có các địa chỉ toán hạng được biết trong thời gian hợp dịch.

Giải pháp của 8051 là dùng kiểu định địa chỉ gián tiếp. Các thanh ghi R0 và R1 có thể hoạt động như là các con trỏ ( pointer ) và nội dung của chúng chỉ ra địa chỉ trong RAM, nơi mà dữ liệu được đọc hay được ghi. Bit có ý nghĩa thấp nhất của opcode ( của lệnh ) xác định thanh ghi nào ( R0 hay R1 ) được sử dụng làm con trỏ ( xem hình 3.1.c ). Trong hợp ngữ của 8051, kiểu định địa chỉ gián tiếp được nhận biết nhờ vào ký tự @ đặt trước R0 hoặc R1. Lấy thí dụ nếu R1 chứa 40H và địa chỉ 40H của bộ nhớ nội chứa 55H, lệnh :

```
MOV A, @R1
```

nạp 55H cho thanh chứa A.

Ta cần đến kiểu định địa chỉ gián tiếp khi ta duyệt các vị trí liên tiếp trong bộ nhớ. Thí dụ sau thực hiện việc tuần tự xóa RAM nội từ địa chỉ 60H đến 7FH :

```
        MOV   R0, #60H
LOOP:   MOV   @R0, #0
        INC   R0
        CJNE  R0, #80H, LOOP
        ( tiếp tục )
```

Lệnh đầu tiên khởi động R0 với nội dung là 60H, địa chỉ bắt đầu của khối nhớ trong RAM; lệnh thứ hai sử dụng kiểu định địa chỉ gián tiếp để nạp 00H cho vị trí được trỏ bởi R0; lệnh thứ ba tăng con trỏ ( R0 ) để trỏ đến địa chỉ tiếp theo và lệnh cuối cùng kiểm tra con trỏ xem đã kết thúc khối nhớ chưa. Việc kiểm tra sử dụng hằng số 80H thay vì là 7FH vì lệnh tăng xuất hiện sau lệnh di chuyển. Điều này đảm bảo vị trí nhớ sau cùng ( 7FH ) được ghi 00H trước khi kết thúc.

### 3.2.4 Định địa chỉ tức thời

Khi toán hạng nguồn là một hằng số thay vì là một biến ( nghĩa là lệnh sử dụng 1 giá trị đã biết trước ở thời gian hợp dịch ), hằng số này

có thể đưa vào lệnh và đây là byte dữ liệu tức thời ( byte thêm vào này có giá trị biết trước ) ( xem hình 3.1.d ).

Trong hợp ngữ, các toán hạng tức thời được nhận biết nhờ vào ký tự # đặt trước chúng. Toán hạng này có thể là một hằng số học, một biến hoặc một biểu thức số học sử dụng các hằng số, các ký hiệu và các toán tử. Trình dịch hợp ngữ tính giá trị và thay thế dữ liệu tức thời vào trong lệnh. Thí dụ lệnh :

```
MOV  A, #12
```

nạp giá trị 12 ( 0CH ) vào thanh chứa A.

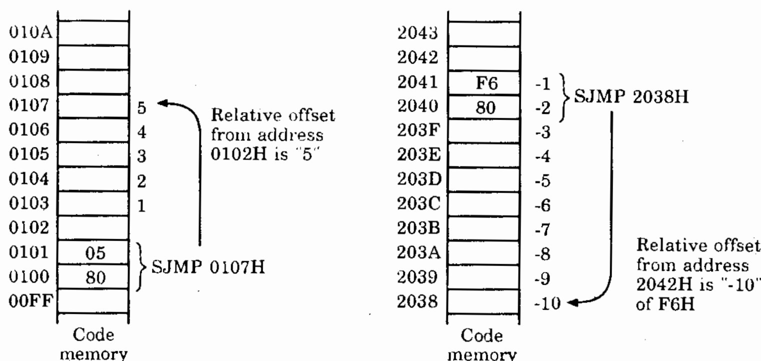
Tất cả các lệnh sử dụng kiểu định địa chỉ tức thời đều sử dụng hằng dữ liệu 8-bit làm dữ liệu tức thời Có một ngoại lệ khi ta khởi động con trỏ dữ liệu 16-bit DPTR, hằng địa chỉ 16-bit được cần đến. Thí dụ :

```
MOV  DPTR, #8000H
```

là một lệnh 3-byte, lệnh này nạp hằng địa chỉ 8000H vào con trỏ dữ liệu DPTR.

### 3.2.5 Định địa chỉ tương đối

Kiểu định địa chỉ tương đối chỉ được sử dụng cho các lệnh nhảy. Một địa chỉ tương đối ( hay còn gọi là *offset* ) là một giá trị 8-bit có dấu. Giá trị này được cộng với bộ đếm chương trình để tạo ra địa chỉ của lệnh tiếp theo cần được thực thi. Do ta sử dụng một *offset* 8-bit có dấu, tầm nhảy được giới hạn là –128 byte đến 127 byte. Byte địa chỉ tương đối là byte thêm vào tiếp theo byte opcode của lệnh ( xem hình 3.1.e ).




**Hình 3.2 : Tính *offset* cho kiểu định địa chỉ tương đối (a) nhảy tới và ngắn (b) nhảy lùi và ngắn**

Short jump ahead in memory : nhảy tới và ngắn

Short jump back in memory : nhảy lùi và ngắn

Relative offset from address 0102H is 5 : *offset* tương đối từ địa chỉ 0102H là 5

Relative offset from address 2042H is –10 ( F6H ) : *offset* tương đối từ địa chỉ 2042H là –10 ( F6H )

Nhờ vào phép cộng, bộ đếm chương trình được tăng đến địa chỉ theo sau lệnh nhảy; vậy thì địa chỉ mới liên quan đến lệnh kế tiếp, không liên quan đến địa chỉ của lệnh nhảy ( xem hình 3.2 ).

Thông thường chi tiết này không liên quan đến người lập trình do bởi các đích nhảy thường được xác định bằng các nhãn và trình dịch hợp ngữ sẽ xác định *offset* tương đối tương ứng. Thí dụ nếu nhãn THERE đặt trước lệnh ở địa chỉ 1040H, lệnh :

SJMP THERE

ở trong bộ nhớ tại địa chỉ 1000H và 1001H, trình dịch hợp ngữ sẽ gán *offset* tương đối là 3EH cho byte 2 của lệnh ( 1002H + 3EH = 1040H )

Định địa chỉ tương đối có điểm lợi là cung cấp cho ta mã không phụ thuộc vào vị trí ( vì các địa chỉ tuyệt đối không được dùng ), nhưng lại có điểm bất lợi là các đích nhảy bị giới hạn trong tầm.

### 3.2.6 Định địa chỉ tuyệt đối

Kiểu định địa chỉ tuyệt đối chỉ được sử dụng với các lệnh ACALL và AJMP. Đây là các lệnh 2-byte cho phép rẽ nhánh chương trình trong trang 2K hiện hành của bộ nhớ chương trình bằng cách cung cấp 11 bit thấp của địa chỉ đích, trong đó 3 bit cao ( A8 – A10 ) đưa vào opcode và 8 bit thấp ( A0 – A7 ) thành lập byte thứ 2 của lệnh. ( xem hình 3.1f ).

5 bit cao của địa chỉ đích là 5 bit cao hiện hành trong bộ đếm chương trình, do vậy lệnh theo sau lệnh rẽ nhánh và đích của lệnh rẽ nhánh phải ở trong cùng 1 trang 2K, bởi vì A11 – A15 không thay đổi ( xem hình 3.3 ). Lấy thí dụ nếu nhãn THERE đặt trước lệnh ở địa chỉ 0F46H, lệnh :

AJMP THERE

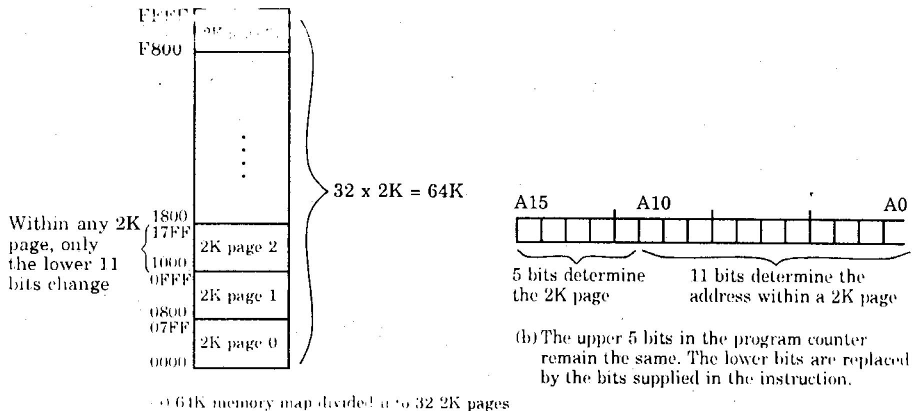
ở trong bộ nhớ tại địa chỉ 0900H và 0901H, trình dịch hợp ngữ sẽ mã hóa lệnh như sau :

| | | |
|---|---|---|
| 11100001 | – | byte 1 ( A10 – A8 + opcode ) |
| 01000110 | – | byte 2 ( A7 – A0 ) |

Các bit được gạch dưới là 11 bit thấp của địa chỉ đích, 0F46H = 0000111101000110B. Năm bit cao ở trong bộ đếm chương trình sẽ không

thay đổi khi lệnh trên được thực thi. Lưu ý là lệnh AJMP và đích nhảy đến đều ở trong 1 trang 2K giới hạn bởi 0800H – 0FFFH (xem hình 3.3), và do vậy có 5 bit địa chỉ cao như nhau.

Địa chỉ tuyệt đối có điểm lợi là lệnh ngắn (. 2-byte ) nhưng có điểm bất lợi là đích bị giới hạn tầm địa chỉ và cung cấp cho ta mã phụ thuộc vào vị trí.




Hình 3.3 : Mã hóa lệnh dùng kiểu định địa chỉ tuyệt đối (a) bộ nhớ được chia thành nhiều trang 2K (b) Bên trong một trang 2K, 5 bit địa chỉ cao không thay đổi.

Within any 2 K page, only the lower 11 bits change : trong một trang 2 K bất kỳ chỉ có 11 bit thấp thay đổi

5 bits determine the 2 K page : 5-bit xác định trang 2 K

11 bits determine the address within a 2 K page : 11-bit xác định địa chỉ trong trang 2 K.

64 K memory map divided into 32 x 2 K pages : bản đồ bộ nhớ 64 K được chia thành 32 trang 2 K.

The upper 5 bits in the program counter remain the same. The lower bits are replaced by the bits supplied in the instruction : 5-bit cao trong thanh ghi PC giữ không đổi. Các bit thấp được thay bằng các bit cung cấp bởi lệnh.

### 3.2.7 Định địa chỉ dài

Kiểu định địa chỉ dài chỉ được dùng cho các lệnh LCALL và LJMP. Các lệnh 3-byte này chứa địa chỉ đích 16-bit ( 2 byte : byte 2 và byte 3 ) của lệnh ( xem hình 3.1.g ).

Lợi ích của kiểu định địa chỉ này là sử dụng hết toàn bộ không gian nhớ chương trình 64K, nhưng lại có điểm bất lợi là lệnh dài đến 3-byte và phụ thuộc vào vị trí.

Việc phụ thuộc vào vị trí được xem là bất lợi do bởi chương trình không thể được thực thi ở một địa chỉ khác. Lấy thí dụ nếu chương trình bắt đầu ở 2000H và có một lệnh như là LJMP 2040H, chương trình này không thể di chuyển đến 4000H chẳng hạn. Lệnh LJMP sẽ vẫn nhảy đến địa chỉ 2040H và đây không phải là vị trí đúng sau khi chương trình đã được di chuyển.

### 3.2.8 Định địa chỉ chỉ số

Kiểu định địa chỉ chỉ số sử dụng một thanh ghi nền ( hoặc bộ đếm chương trình hoặc con trỏ dữ liệu ) và một *offset* ( thanh chứa A ) tạo thành dạng địa chỉ hiệu dụng cho lệnh JMP hoặc MOVC ( xem hình 3.1.h ). Trong nhiều ứng dụng, các bảng nhảy hoặc các bảng tìm kiếm được tạo ra dễ dàng bằng cách sử dụng kiểu định địa chỉ chỉ số. Các thí dụ được cho ở phụ lục C với các lệnh MOVC A, @A+<base-reg> và JMP @A+DPTR

## 3.3 CÁC LOẠI LỆNH

Các lệnh của 8051 được chia làm 5 nhóm :

- Nhóm lệnh số học
- Nhóm lệnh logic
- Nhóm lệnh di chuyển dữ liệu
- Nhóm lệnh xử lý bit
- Nhóm lệnh rẽ nhánh

Phụ lục A giúp ta tham chiếu nhanh tất cả các lệnh của 8051 đã được phân nhóm.

### 3.3.1 Các lệnh số học

Các lệnh số học cũng được phân loại như trong phụ lục A. Do có thể có 4 khả năng định địa chỉ, lệnh ADD A còn được viết dưới các dạng sau :

| | |
|---|---|
| ADD A, 7FH | [định địa chỉ trực tiếp] |
| ADD A, @R0 | [định địa chỉ gián tiếp] |
| ADD A, R7 | [định địa chỉ thanh ghi] |
| ADD A, #35H | [định địa chỉ tức thời] |

Tất cả các lệnh số học được thực thi trong một chu kỳ máy ngoại trừ lệnh INC DPTR được thực thi trong hai chu kỳ máy, các lệnh MUL

AB và DIV AB thực thi trong 4 chu kỳ máy ( lưu ý là một chu kỳ máy dài 1 μsec nếu 8051 hoạt động với xung clock 12 MHz ).

8051 cung cấp kiểu định địa chỉ linh hoạt cho không gian nhớ nội. Một vị trí bất kỳ đều có thể tăng hay giảm bằng cách dùng kiểu định địa chỉ trực tiếp mà không cần qua trung gian thanh chứa A. Lấy thí dụ nếu vị trí 7FH của RAM nội chứa giá trị 40H, lệnh :

```
INC 7FH
```

tăng giá trị tại địa chỉ 7FH thành 41H và đặt kết quả tại 7FH.

8051 cũng có lệnh INC thao tác trên con trỏ dữ liệu 16-bit. Do con trỏ dữ liệu chứa 16-bit địa chỉ của bộ nhớ ngoài, việc tăng nội dung con trỏ này là một đặc trưng thường sử dụng. Không may cho ta là không có lệnh giảm nội dung con trỏ dữ liệu và ta phải dùng đến một chuỗi lệnh sau thay cho điều thiếu sót này :

```
        DEC  DPL                ; giảm byte thấp của DPTR
        MOV  R7, DPL            ; cất vào R7
        CJNE R7, #0FFH, SKIP    ; so sánh với 0FFH ?
        DEC DPH                 ; bằng : giảm byte cao của DPTR
SKIP :  ( tiếp tục )            ; không bằng : bỏ qua
```

Các byte thấp và byte cao của DPTR phải được giảm riêng rẽ trong đó byte cao ( DPH ) chỉ được giảm 1 nếu byte thấp ( DPL ) tràn từ 00H qua FFH.

Lệnh nhân MUL AB nhân dữ liệu 8-bit chứa trong thanh ghi B với nội dung của thanh chứa A và đặt kết quả 16-bit trong cặp thanh ghi BA ( thanh ghi B chứa byte cao, thanh chứa A chứa byte thấp ).

Lệnh DIV AB chia nội dung thanh ghi A cho dữ liệu chứa trong thanh ghi B, thương số 8 bit cất trong thanh chứa A và dư số 8-bit cất trong thanh ghi B. Lấy thí dụ nếu nội dung của thanh chứa là 25 ( 19H ) và thanh ghi B chứa 6 ( 06H ), lệnh sau :

```
DIV AB
```

chia 25 cho 6, kết quả 4 cất trong thanh chứa A và dư số 1 cất trong thanh ghi B.

Với số BCD, các lệnh ADD và ADDC phải được tiếp theo bởi lệnh DA A để đảm bảo rằng kết quả ở trong tầm số BCD. Lưu ý là lệnh hiệu đính DA A sẽ không biến đổi số nhị phân thành BCD mà chỉ tạo ra 1 kết quả hợp lệ và ta thường gọi là hiệu đính trong phép cộng 2 số BCD.

Lấy thí dụ nếu thanh chứa A chứa giá trị BCD là 59 (59H), chuỗi lệnh sau :

```
ADD  A, #1
DA   A
```

trước tiên cộng 1 với nội dung thanh chứa A, kết quả là 5AH. Kết quả này được lệnh thứ hai hiệu đính thành giá trị BCD hợp lệ là 60 ( 60H ) và cất vào thanh chứa A.

### 3.3.2 Nhóm lệnh logic

Nhóm lệnh logic của 8051 thực hiện các phép toán logic ( AND, OR, XOR và NOT ) trên các byte dữ liệu và thực hiện trên từng bit có cùng giá trị vị trí ( trọng số ). Nếu thanh chứa A chứa giá trị 00110101B, lệnh AND logic sau đây :

```
AND  A, #01010011B
```

sẽ tạo ra kết quả là 00010001B cất trong thanh chứa A. Điều này được minh họa như sau :

| | | |
|---|---|---|
| | 01010011 | ( dữ liệu tức thời ) |
| AND | 00110101 | ( giá trị ban đầu chứa trong A ) |
| | 00010001 | ( kết quả chứa trong A ) |

Các kiểu định địa chỉ cho các lệnh logic cũng giống như các kiểu định địa chỉ cho các lệnh số học, lệnh AND logic có thể có các dạng sau :

| | |
|---|---|
| AND A, 55H | ( định địa chỉ trực tiếp ) |
| AND A, @R0 | ( định địa chỉ gián tiếp ) |
| AND A, R6 | ( định địa chỉ thanh ghi ) |
| AND A, #33H | ( định địa chỉ tức thời ) |

Tất cả các lệnh logic sử dụng thanh chứa A để lưu một toán hạng sẽ được thực thi trong 1 chu kỳ máy, ngược lại nếu sử dụng thanh ghi khác hoặc byte nhớ thay cho thanh chứa A, lệnh phải được thực thi trong 2 chu kỳ máy.

Các phép toán logic có thể được thực hiện trên một byte bất kỳ trong bộ nhớ dữ liệu nội mà không cần qua trung gian thanh chứa A. Lệnh " XRL direct, #data " giúp ta nhanh chóng và dễ dàng đảo mức logic các bit của *port*. Thí dụ lệnh sau :

```
XRL  P1, #0FFH
```

thực hiện một thao tác đọc – sửa – ghi. 8 bit của *port* 1 được đọc, sau đó từng bit được XOR với các bit tương ứng ( cùng vị trí ) của dữ liệu tức thời. Vì 8 bit của dữ liệu tức thời đều là 1, kết quả là từng bit của *port* 1 được lấy bù ( vì $A \oplus 1 = \overline{A}$ ). Kết quả này được ghi trở lại *port* 1.

Các lệnh quay ( RL A và RR A ) dịch thanh chứa A qua trái hoặc qua phải 1-bit. Với lệnh quay trái ( RL A ), bit có giá trị vị trí lớn nhất MSB được đưa vào vị trí có giá trị thấp nhất LSB. Với lệnh quay phải ( RR A ), bit có giá trị vị trí thấp nhất LSB được đưa vào vị trí có giá trị lớn nhất MSB. Các lệnh RLC A và RRC A là các lệnh quay 9-bit sử dụng thanh chứa A và cờ nhớ CY trong thanh ghi PSW. Thí dụ nếu cờ nhớ CY chứa 1 và thanh chứa A chứa 00H, lệnh sau :

```
RRC A
```

sẽ cho kết quả là : cờ nhớ CY chứa 0 và thanh chứa A có nội dung là 80H. Điều này có nghĩa là cờ nhớ CY được đưa đến ACC.7 và ACC.0 được đưa đến cờ nhớ.

Lệnh SWAP A trao đổi nửa thấp 4-bit với nửa cao 4-bit trong thanh chứa A với nhau. Lệnh này thường dùng trong các phép toán số BCD. Lấy thí dụ nếu thanh chứa A chứa một số nhị phân đã biết và có giá trị nhỏ hơn $100_{(10)}$, ta có thể biến đổi số nhị phân này thành số BCD bằng các dòng lệnh như sau :

```
MOV B, #10
DIV AB
SWAP A
ADD A, B
```

Việc chia một số cho 10 trong hai lệnh đầu tiên tạo ra digit chục trong nửa thấp của thanh chứa A và digit đơn vị trong thanh ghi B. Lệnh SWAP và ADD di chuyển digit chục đến nửa cao của thanh chứa A và digit đơn vị vào nửa thấp của thanh chứa này.

### 3.3.3 Các lệnh di chuyển dữ liệu

#### Trong RAM nội

Các lệnh di chuyển dữ liệu bên trong không gian nhớ nội được thực thi trong 1 hay 2 chu kỳ máy. Dạng của lệnh như sau :

```
MOV <destination>, <source>
```

Lệnh trên cho phép dữ liệu được di chuyển giữa các vị trí của RAM nội hoặc các thanh ghi chức năng đặc biệt SFR mà không cần qua trung gian thanh chứa A. Cần nhớ là 128 byte cao của RAM dữ liệu ( đối với

8032/8052 ) chỉ được truy xuất bằng kiểu định địa chỉ gián tiếp và các thanh ghi chức năng đặc biệt chỉ được truy xuất bằng kiểu định địa chỉ trực tiếp.

Một đặc trưng làm cho cấu trúc của MCS-51 khác với cấu trúc của hầu hết các bộ vi xử lý là vùng *stack* thường trú trong RAM trên *chip* ( RAM nội ) và tăng dần về phía trên của bộ nhớ, phía các địa chỉ cao hơn. Lệnh PUSH trước tiên tăng con trỏ *stack* ( SP ) rồi sao chép byte vào trong *stack*. Các lệnh PUSH và POP sử dụng kiểu định địa chỉ trực tiếp để nhận biết byte được cất hoặc được phục hồi nhưng bản thân *stack* được truy xuất bởi kiểu định địa chỉ gián tiếp sử dụng con trỏ *stack* SP.

Điều này có nghĩa là vùng *satck* có thể sử dụng 128 byte cao của bộ nhớ RAM nội trên 8032/8052. 128 byte cao này không có trong 8031/8051. Với các bộ vi điều khiển này, nếu nội dung của SP vượt quá 7FH ( 127 ) ( nghĩa là nội dung của SP lớn hơn 7FH ), các byte được PUSH sẽ bị mất còn các byte được POP không được xác định.

Các lệnh chuyển dữ liệu còn bao gồm lệnh MOV 16-bit dùng để khởi động con trỏ dữ liệu DPTR cho mục đích tìm kiếm các bảng trong bộ nhớ chương trình hoặc cho mục đích truy xuất bộ nhớ dữ liệu ngoài 16-bit.

Lệnh hoán đổi nội dung XCH được sử dụng để hoán đổi nội dung của thanh chứa A với nội dung của byte được chỉ ra trong lệnh. Dạng lệnh như sau :

XCH A, <source>

Lệnh trên làm cho thanh chứa A và byte được định địa chỉ trao đổi dữ liệu với nhau. Việc trao đổi một digit sử dụng lệnh có dạng :

XCHD A, @Ri

cũng hoạt động tương tự, tuy nhiên chỉ có các nửa thấp của các byte được trao đổi với nhau. Thí dụ nếu thanh chứa A chứa F3H, R1 chứa 40H và tại địa chỉ 40H trong RAM nội chứa 5BH, lệnh :

XCHD A, @R1

cho kết quả là A chứa FBH và tại địa chỉ 40H trong RAM nội chứa 53H.

**Trong RAM ngoài**

Với các lệnh mà việc di chuyển dữ liệu cho phép dữ liệu được di chuyển giữa RAM nội với RAM ngoài, ta phải sử dụng kiểu định địa chỉ gián tiếp. Các địa chỉ gián tiếp được xác định bằng cách dùng địa chỉ 1-byte ( như @Ri, trong đó Ri là R0 hoặc R1 của dãy thanh ghi được chọn )

hoặc địa chỉ 2-byte ( như @DPTR ). Điểm bất lợi khi dùng địa chỉ 16-bit là tất cả 8 bit của *port* 2 phải được dùng như byte cao của bus địa chỉ và điều này sẽ không cho ta sử dụng *port* 2 làm *port* xuất/nhập. Ngược lại các địa chỉ 8-bit cho phép ta truy xuất đến một vài KB của RAM mà không cần sử dụng toàn bộ *port* 2 ( xem mục " Truy xuất bộ nhớ dữ liệu ngoài ở chương 2 ).

Tất cả các lệnh di chuyển dữ liệu hoạt động trên bộ nhớ ngoài được thực thi trong 2 chu kỳ máy và sử dụng thanh chứa làm toán hạng nguồn hoặc toán hạng đích.

Các tín hiệu dùng để truy xuất RAM ngoài ( $\overline{RD}$ và $\overline{WR}$ ) chỉ tích cực trong khi lệnh MOVX được thực thi. Bình thường các tín hiệu này không tích cực ( mức cao ) và nếu bộ nhớ ngoài không được sử dụng, các đường $\overline{RD}$ và $\overline{WR}$ có chức năng như các đường xuất / nhập.

**Các bảng tìm kiếm**

Có hai lệnh di chuyển dữ liệu dành cho việc đọc các bảng tìm kiếm trong bộ nhớ chương trình. Do bởi các lệnh này truy xuất bộ nhớ chương trình, các bảng tìm kiếm chỉ có thể được đọc và không được cập nhật. Mã gợi nhớ của lệnh là MOVC ( move constant : di chuyển hằng ). MOVC sử dụng hoặc bộ đếm chương trình hoặc con trỏ dữ liệu làm thanh ghi nền và thanh chứa A chứa địa chỉ *offset*. Lệnh sau :

```
MOVC   A, @A+DPTR
```

có thể truy xuất một bảng 256 điểm nhập được đánh số từ 0 đến 255. Số của điểm nhập yêu cầu được nạp cho thanh chứa A và con trỏ dữ liệu được khởi động để chứa địa chỉ đầu bảng. Lệnh sau :

```
MOVC   A, @A+PC
```

cũng hoạt động tương tự, ngoại trừ ở đây bộ đếm chương trình được dùng để chứa địa chỉ nền và bảng được truy xuất nhờ vào một chương trình con. Trước tiên số của điểm nhập yêu cầu được nạp cho thanh chứa A, sau đó chương trình con được gọi. Chuỗi lệnh cho phép khởi động và gọi có thể là :

```
                MOV   A, ENTRY-NUMBER
                CALL  LOOK-UP
                .
                .
                .
LOOK-UP:        INC  A
                MOVC   A, @A+PC
```

```
            RET
TABLE :     DB    data, data, data, ....
```

Bảng được định nghĩa ngay sau lệnh RET trong chương trình. Lệnh tăng được cần đến do PC trỏ tới lệnh RET khi lệnh MOVC được thực thi. Việc tăng nội dung thanh chứa sẽ bỏ qua lệnh RET..

### 3.3.4 Các lệnh xử lý bit

Bộ xử lý của 8051 chứa 1 bộ xử lý logic trên bit cho phép ta thực hiện các phép toán đơn bit. RAM nội chứa 128 bit được định địa chỉ và không gian SFR hỗ trợ thêm đến 128 bit được định địa chỉ. Tất cả các đường *port* đều có địa chỉ bit và mỗi đường có thể được xử lý như là một *port* đơn bit riêng rẽ. Các lệnh truy xuất các bit này không chỉ là các lệnh rẽ nhánh có điều kiện mà còn là các lệnh di chuyển, *set*, xóa, lấy bù, OR và AND. Các thao tác trên bit như vậy ( một trong các đặc trưng mạnh của MCS-51 ) không dễ dàng có được trong các cấu trúc khác, các cấu trúc sử dụng các thao tác hướng byte.

Mọi thao tác truy xuất bit đều sử dụng kiểu định địa chỉ trực tiếp với các địa chỉ bit từ 00H đến 7FH trong 128 vị trí thấp, và từ địa chỉ 80H đến FFH trong không gian SFR. Các địa chỉ bit ở 128 vị trí thấp thuộc các byte có địa chỉ từ 20H đến 2FH được đánh số liên tục từ bit 0 của byte ở địa chỉ 20H (bit 00H) đến bit 7 của byte ở địa chỉ 2FH (bit 7FH).

Các bit có thể được *set* và xóa bằng 1 lệnh. Điều khiển đơn bit được dùng cho nhiều thiết bị xuất/nhập, bao gồm xuất ra rờ le, động cơ, cuộn dây, các LED, mạch còi báo động, loa hoặc nhập từ các chuyển mạch hoặc các bộ chỉ thị trạng thái. Nếu có một mạch còi báo động nối với bit 7 của *port* 1, ta có thể tác động mạch còi bằng cách *set* bit của *port* :

```
SETB   P1.7
```

hoặc tắt còi bằng cách xóa bit của *port* :

```
CLR   P1.7
```

Trình dịch hợp ngữ sẽ biến đổi ký hiệu P1.7 thành địa chỉ bit là 97H. Thí dụ sau cho phép ta di chuyển 1 cờ vào một chân của *port* :

```
MOV   C, FLAG
MOV   P1.0, C
```

Trong thí dụ trên, FLAG là tên của 1 bit được định địa chỉ trong 128 vị trí thấp hoặc trong không gian SFR. Một đường xuất/nhập ( ở thí dụ trên là bit 0 của *port* 1 ) được *set* hoặc xóa phụ thuộc vào bit cờ có giá trị 1 hay 0. Bit nhớ trong PSW được dùng như một thanh chứa đơn bit

của bộ xử lý logic trên bit, các lệnh đơn bit liên quan đến bit nhớ ký hiệu là C là các lệnh đặc biệt liên quan đến cờ nhớ ( như là CLR C ). Bit nhớ cũng có một địa chỉ trực tiếp vì bit này được lưu trong thanh ghi PSW, thanh ghi này được định địa chỉ từng bit.

Cũng giống như các thanh ghi khác được định địa chỉ từng bit trong không gian SFR, các bit của PSW có mã gợi nhớ mà trình dịch hợp ngữ sẽ chấp nhận thay cho địa chỉ bit. Mã gợi nhớ của cờ nhớ là CY được định nghĩa thay cho địa chỉ bit 0D7H. Ta hãy khảo sát 2 lệnh sau :

```
CLR  C
CLR  CY
```

Cả 2 đều có cùng công dụng, tuy nhiên dạng lệnh trước là lệnh 1-byte trong khi dạng lệnh sau là lệnh 2-byte. Trong dạng lệnh sau byte thứ 2 là địa chỉ trực tiếp của bit được xác định – cờ nhớ.

Các lệnh logic trên bit bao gồm cả lệnh ANL và ORL nhưng không bao gồm lệnh XRL. Nếu ta cần XOR 2 bit, BIT1 và BIT2, và kết quả cất trong cờ nhớ, các lệnh sau được sử dụng :

```
        MOV   C, BIT1
        JNB   BIT2, SKIP
        CPL   C
SKIP :
```

Trước tiên BIT1 được nạp cho cờ nhớ. Nếu BIT2 = 0, thì C đã chứa kết quả ( nghĩa là BIT1 ⊕ BIT2 = BIT1 nếu BIT2 = 0 ). Nếu BIT2 = 1, C chứa kết quả là bù của cờ nhớ. Việc lấy bù C hoàn tất phép toán XOR.

Chương trình trong thí dụ trên sử dụng lệnh JNB, một trong chuỗi lệnh kiểm tra bit. Các lệnh này sẽ nhảy nếu bit được định địa chỉ được *set* bằng 1 ( như là lệnh JC, JB, JBC ) hoặc nếu bit được định địa chỉ không được *set* ( JNC, JNB ). Trong thí dụ ở trên nếu BIT2 = 0, lệnh CPL C được bỏ qua. Lệnh JBC ( nhảy nếu bit được *set*, sau đó xóa bit ) thực hiện việc nhảy nếu bit được định địa chỉ được *set* và cũng xóa bit; vậy thì một cờ có thể được kiểm tra và xóa bằng 1 lệnh.

Tất cả các bit của PSW đều có thể định địa chỉ trực tiếp, do vậy bit chẵn lẻ hoặc các cờ đa mục đích cũng hợp lệ đối với các lệnh kiểm tra bit.

### 3.3.5 Các lệnh rẽ nhánh

Trong tập lệnh của 8051 có nhiều lệnh điều khiển luồng chương trình, bao gồm các lệnh gọi một thủ tục và quay về từ một thủ tục, rẽ

nhánh có điều kiện hoặc không có điều kiện. Các khả năng này được cải tiến hơn nữa bởi 3 kiểu định địa chỉ cho các lệnh rẽ nhánh chương trình.

Có 3 biến thể của lệnh nhảy : SJMP, LJMP và AJMP ( sử dụng kiểu định địa chỉ tương đối, dài và tuyệt đối ).

Trình dịch hợp ngữ của Intel ( ASM51 ) cho phép sử dụng mã gợi nhớ JMP nếu người lập trình không quan tâm đến các biến thể. Trình dịch hợp ngữ của các công ty khác có thể không hỗ trợ đặc tính này. JMP tổng quát dịch thành AJMP nếu đích nhảy đến không chứa tham chiếu thuận và ở trong một trang 2K. Ngược lại, JMP được dịch thành LJMP. Lệnh CALL cũng hoạt động theo cách này.

Lệnh SJMP xác định địa chỉ đích là *offset* tương đối như đã bàn đến trước đây khi đề cập đến các kiểu định địa chỉ. Vì lệnh dài 2-byte ( bao gồm một opcode cộng với một *offset* tương đối 8-bit ), khoảng cách nhảy được giới hạn từ −128 byte đến +127 byte so với địa chỉ của lệnh theo sau lệnh SJMP.

Lệnh LJMP xác định địa chỉ đích là hằng số 16-bit. Vì lệnh dài 3-byte ( bao gồm 1 opcode cộng với 2-byte địa chỉ ), địa chỉ đích có thể ở bất cứ đâu trong không gian nhớ chương trình 64K.

Lệnh AJMP xác định địa chỉ đích là một hằng số 11-bit. Cũng như lệnh SJMP, lệnh AJMP cũng dài 2 byte nhưng được mã hóa khác. Byte opcode sẽ chứa 3 trong 11 bit địa chỉ và byte 2 chứa 8 bit thấp của địa chỉ đích. Khi lệnh được thực thi, 11 bit này thay chỗ cho 11 bit thấp trong PC còn 5 bit cao của PC vẫn giữ nguyên. Địa chỉ đích do vậy phải ở trong cùng một trang 2 K với lệnh theo sau lệnh AJMP. Do ta có không gian nhớ chương trình là 64 K, ta có 32 trang và mỗi trang bắt đầu ở địa chỉ là biên của 2 K ( như là 0000H, 0800H, 1000H, 1800H, ... cho đến F800H : xem hình 3.3 )

Trong mọi trường hợp người lập trình xác định địa chỉ đích cho trình dịch hợp ngữ theo cách thông thường như là nhãn hoặc là 1 hằng số 16-bit. Trình dịch hợp ngữ sẽ đặt địa chỉ đích theo đúng khuôn dạng của từng lệnh. Nếu khuôn dạng yêu cầu bởi lệnh không được hỗ trợ khoảng cách dùng để xác định địa chỉ đích, một thông báo " địa chỉ đích ngoài tầm " sẽ được đưa ra.

**Các bảng nhảy**

Lệnh JMP @A+DPTR hỗ trợ các thao tác nhảy phụ thuộc vào trường hợp cụ thể cho các bảng nhảy. Địa chỉ đích được tính ở thời điểm thực thi lệnh là tổng của nội dung thanh ghi DPTR 16-bit với nội dung

của thanh chứa A. DPTR được nạp địa chỉ của bảng nhảy và thanh chứa A đóng vai trò của một thanh ghi chỉ số. Thí dụ nếu có 5 trường hợp được yêu cầu, một giá trị từ 0 đến 4 được nạp cho thanh chứa A và một nhảy cho trường hợp tương thích được thực hiện như sau :

```
MOV     DPTR, #JUMP_TABLE
MOV     A, INDEX_NUMBER
RL      A
JMP     @A+DPTR
```

Lệnh RL A ở trên biến đổi chỉ số ( 0 → 4 ) thành 1 số chẵn trong tầm từ 0 đến 8 vì mỗi điểm nhập trong bảng nhảy là 1 địa chỉ 2-byte :

```
JUMP_TABLE :        AJMP   CASE0
                    AJMP   CASE1
                    AJMP   CASE2
                    AJMP   CASE3
```

### Chương trình con và ngắt

Có 2 biến thể cho lệnh CALL : ACALL và LCALL sử dụng kiểu định địa chỉ tuyệt đối và dài. Cũng như lệnh JMP, mã gợi nhớ CALL có thể được sử dụng với trình dịch hợp ngữ của Intel nếu người lập trình không quan tâm đến cách định địa chỉ. Cả 2 lệnh trên đều cất nội dung của bộ đếm chương trình vào *stack* và nạp cho bộ đếm chương trình địa chỉ đã được xác định trong lệnh. Lưu ý là PC sẽ chứa địa chỉ của lệnh theo sau lệnh CALL khi nội dung thanh ghi này được cất vào *stack*. Khi nạp vào *stack*, byte thấp nạp trước và byte cao nạp sau. Các byte được lấy ra từ *stack* theo trình tự ngược lại. Lấy thí dụ nếu lệnh LCALL được chứa trong bộ nhớ chương trình ở các địa chỉ là 1000H-1002H và con trỏ *stack* chứa 20H, lệnh LCALL sẽ :

(a) nạp địa chỉ quay về 1003H vào *stack* ( đặt 03H tại địa chỉ 21H và 10H tại địa chỉ 22H ).

(b) nội dung của con trỏ *stack* bây giờ là 22H

(c) nhảy đến chương trình con bằng cách nạp cho PC địa chỉ chứa trong byte 2 và byte 3 của lệnh.

Các lệnh LCALL và ACALL cũng có các hạn chế trên địa chỉ đích như các lệnh LJMP và AJMP ( đã đề cập ở trên ).

Các thủ tục cần được kết thúc bằng lệnh RET, lệnh này trả việc thực thi chương trình trở về lệnh theo sau lệnh CALL. Không có điều gì bí ẩn về cách mà lệnh RET trả điều khiển về cho chương trình chính. Đơn

giản là lệnh này lấy lại ( pop ) 2 byte sau cùng ra khỏi *stack* và nạp chúng cho bộ đếm chương trình. Một qui luật chủ yếu cho việc lập trình với các thủ tục là chúng luôn luôn được gọi bởi lệnh CALL và luôn luôn trả điều khiển về chương trình gọi bởi lệnh RET. Các thao tác nhảy vào hoặc nhảy ra khỏi 1 thủ tục bằng 1 cách khác nào đó thường làm rối vùng *stack* và làm cho chương trình bị dừng.

Lệnh RETI trả điều khiển về chương trình gọi từ 1 trình phục vụ ngắt ( ISR : interrupt service routine ). Điểm khác nhau giữa RET và RETI là RETI báo hiệu cho hệ thống điều khiển ngắt rằng quá trình xử lý ngắt đã xong. Nếu không có một ngắt nào duy trì trong thời gian RETI được thực thi, RETI hoạt động giống RET. Các ngắt và lệnh RETI sẽ được đề cập chi tiết trong chương 6.

**Nhảy có điều kiện**

8051 cung cấp cho ta nhiều lệnh nhảy có điều kiện. Tất cả các lệnh này xác định địa chỉ đích bằng kiểu định địa chỉ tương đối và cùng bị giới hạn ở khoảng cách nhảy từ −128 byte đến +127 byte kể từ lệnh theo sau lệnh nhảy có điều kiện. Tuy nhiên cần lưu ý là người lập trình sẽ xác định địa chỉ đích theo cùng cách với các lệnh nhảy khác bằng cách dùng nhãn hoặc bằng số 16-bit. Trình dịch hợp ngữ sẽ thực hiện các việc còn lại. Không có bit 0 trong PSW. Các lệnh JZ và JNZ kiểm tra dữ liệu trong thanh chứa cho điều kiện này.

Lệnh DJNZ ( giảm và nhảy nếu khác không ) dành cho điều khiển lặp vòng. Để thực thi 1 vòng lặp N lần, ta nạp một byte số đếm N cho một thanh ghi và kết thúc vòng lặp với DJNZ trỏ tới điểm bắt đầu vòng lặp. Thí dụ dưới đây có N=10 :

```
        MOV   R7,#10
LOOP:   ( bắt đầu vòng lặp )
        .
        .
        ( kết thúc vòng lặp )
        DJNZ  R7, LOOP
        ( tiếp tục )
```

Lệnh CJNE ( so sánh và nhảy nếu không bằng ) cũng dành cho việc điều khiển vòng lặp. Hai byte được xác định trong trường toán hạng của lệnh và việc nhảy chỉ được thực thi nếu 2 byte khác 0. Thí dụ nếu 1 ký tự vừa được đọc vào thanh chứa A từ *port* nối tiếp và ta muốn nhảy đến

1 lệnh được nhận biết bởi nhãn TERMINATE nếu ký tự là Control-C ( 03H ), các dòng lệnh sau được sử dụng :

```
        CJNE   A, #03H, SKIP
        SJMP   TERMINATE
SKIP :
```

Vì thao tác nhảy chỉ xuất hiện nếu thanh chứa A chứa mã của Control-C, một nhãn SKIP ( bỏ qua ) được dùng để bỏ qua việc kết thúc lệnh nhảy ngoại trừ khi mã yêu cầu được đọc. Lệnh trên còn được ứng dụng trong các phép so sánh lớn hơn hay nhỏ hơn. Hai byte trong trường toán hạng là các số nguyên không dấu. Nếu byte đầu nhỏ hơn byte thứ hai, cờ nhớ được *set* lên 1. Nếu byte đầu lớn hơn hoặc bằng byte thứ hai, cờ nhớ được xóa. Lấy thí dụ nếu ta muốn nhảy đến BIG nếu giá trị trong thanh chứa A lớn hơn hoặc bằng 20H, các chỉ thị sau được dùng :

```
CJNE   A, #20H, $+3
JNC    BIG
```

Đích nhảy cho lệnh CJNE được xác định là $+3. Dấu $ là 1 ký hiệu của trình dịch hợp ngữ biểu thị địa chỉ của lệnh hiện hành. Vì CJNE là lệnh 3-byte, $+3 là địa chỉ của lệnh tiếp theo, JNC. Mặt khác, lệnh CJNE theo sau bởi lệnh JNC không quan tâm đến kết quả so sánh. Mục đích duy nhất của việc so sánh là để *set* hay xóa cờ nhớ và lệnh JNC quyết định nhảy hay không nhảy. Thí dụ này là một thể hiện trong đó 8051 tiếp cận với 1 tình huống lập trình tổng quát một cách vụng về hơn so với hầu hết các bộ vi xử lý, tuy nhiên, như sẽ được trình bày trong chương 7, việc sử dụng các *macro* cho phép ta có các chuỗi lệnh mạnh, như thí dụ trên chẳng hạn, được cấu trúc và thực thi bằng cách dùng một mã gợi nhớ duy nhất.

# 4

# HOẠT ĐỘNG ĐỊNH THỜI

## 4.1 MỞ ĐẦU

Nội dung của chương này khảo sát các bộ định thời ( timer ) của *chip* 8051. Ta hãy bắt đầu từ quan điểm đơn giản về các bộ định thời thường được sử dụng cho các bộ vi xử lý hoặc các bộ vi điều khiển.

Một bộ định thời là một chuỗi các flipflop với mỗi flipflop là một mạch chia 2, chuỗi này nhận một tín hiệu ngõ vào làm nguồn xung *clock*. Xung *clock* đặt vào flipflop thứ nhất, flipflop này chia đôi tần số xung *clock*. Ngõ ra của flipflop thứ nhất trở thành nguồn xung *clock* cho flipflop thứ hai, nguồn xung *clock* này cũng được chia cho 2, v.v... Vì mỗi một tầng kế tiếp nhau đều chia cho 2 nên một bộ định thời có n tầng sẽ chia tần số xung *clock* ở ngõ vào của bộ này cho $2^n$.

Ngõ ra của tầng cuối cùng làm xung *clock* cho một flipflop báo tràn bộ định thời hay còn gọi là cờ tràn ( overflow flag ), cờ tràn này được kiểm tra bởi phần mềm hoặc tạo ra một ngắt. Giá trị nhị phân trong các flipflop của bộ định thời là số đếm của các xung *clock* từ khi bộ định thời bắt đầu đếm. Thí dụ một bộ định thời 16-bit sẽ đếm từ 0000H đến FFFFH. Cờ tràn được *set* bằng 1 khi xảy ra tràn số đếm từ FFFFH xuống 0000H.

Hoạt động của một bộ định thời đơn giản được minh họa trong hình 4.1, bộ định thời 3-bit. Mỗi một tầng là một D.FF kích khởi cạnh âm hoạt động như một mạch chia cho 2 do ta nối ngõ ra $\overline{Q}$ với ngõ vào D. Flipflop cờ đơn giản là một mạch chốt D được *set* bằng 1 bởi tầng cuối của bộ định thời. Giản đồ thời gian ở hình 4.1.b cho thấy tầng thứ nhất ($Q_0$) chia 2 tần số xung *clock*, tầng thứ hai chia 4 tần số xung *clock* và v.v... Số đếm ( count ) được ghi ở dạng thập phân và được kiểm tra dễ dàng bằng cách khảo sát trạng thái của 3 flipflop. Thí dụ số đếm là 4 xuất hiện khi $Q_2 = 1$, $Q_1 = 0$ và $Q_0 = 0$ ( $4_{10} = 100_2$ ). Các flipflop ở hình 4.1 là các flipflop tác động cạnh âm ( nghĩa là ngõ ra Q của các flipflop đổi trạng thái theo cạnh âm của xung *clock* ). Khi số đếm tràn từ $111_2$

xuống $000_2$, ngõ ra $Q_2$ có cạnh âm ( 1 → 0 ) làm cho trạng thái của flip-flop cờ đổi từ 0 lên 1 ( ngõ vào D của flipflop này luôn luôn ở logic 1 ).

Bộ định thời được sử dụng trong hầu hết các ứng dụng hướng điều khiển và 8051 với các bộ định thời trên *chip* không phải là trường hợp ngoại lệ. 8051 có hai bộ định thời 16-bit, mỗi bộ có bốn chế độ hoạt động. Bộ định thời thứ ba với ba chế độ hoạt động được thêm vào đối với *chip* 8052. Các bộ định thời được dùng để :

(a) Định thời trong một khoảng thời gian.

(b) Đếm sự kiện.

(c) Tạo tốc độ baud cho *port* nối tiếp của *chip* 8051.

Timer flip flops (3)
"Flag" flip-flop
Clock
D
$\overline{Q}$
$Q_0$
$Q_1$
$Q_2$
Q
LSB
MSB
"Flag"

Clock
$Q_0$ (LSB)
$Q_1$
$Q_2$ (MSB)
Count 0 1 2 3 4 5 6 7 0
Flag
Flag is set on 7-to-0 timer overflow

Hình 4.1 : Một bộ định thời 3-bit (a) sơ đồ logic (b) giản đồ thời gian

Timer flipflops : các flipflop định thời
" Flag " flipflop : flipflop cờ
Count : số đếm
Flag is set on 7-to-0 timer overflow : cờ được *set* khi có tràn bộ định thời ( số đếm tràn từ 7 xuống 0 )

Với bộ định thời 16-bit, tầng cuối cùng ( tầng thứ 16 ) chia tần số xung *clock* ở ngõ vào của bộ định thời cho $2^{16} = 65536$.

Trong các ứng dụng định thời trong một khoảng thời gian, bộ định thời được lập trình sao cho sẽ tràn sau 1 khoảng thời gian qui định và *set* cờ tràn của bộ định thời bằng 1. Cờ tràn được sử dụng để đồng bộ chương trình nhằm thực hiện một công việc như là kiểm tra trạng thái của các ngõ nhập hoặc gởi dữ liệu đến các ngõ xuất. Các ứng dụng khác có thể sử dụng xung *clock* qui định của bộ định thời để đo khoảng thời gian giữa 2 sự kiện ( thí dụ đo độ rộng xung ).

Việc đếm sự kiện được dùng để xác định số lần xuất hiện của một sự kiện hơn là đo thời gian giữa các sự kiện. Từ "sự kiện" là một kích thích bên ngoài cung cấp một chuyển trạng thái từ 1 xuống 0 tới một chân của *chip* 8051. Các bộ định thời cũng có thể cung cấp xung *clock* tốc độ baud cho *port* nối tiếp bên trong 8051.

Các bộ định thời của 8051 được truy xuất bằng cách sử dụng 6 thanh ghi chức năng đặc biệt ( xem bảng 4.1 ). Với bộ định thời thứ ba của *chip* 8052, ta có thêm 5 thanh ghi chức năng đặc biệt nữa để truy xuất bộ định thời này.

| ***SFR của bộ định thời*** | ***Mục đích*** | ***Địa chỉ*** | ***Định địa chỉ bit*** |
|---|---|---|---|
| TCON | Điều khiển | 88H | Có |
| TMOD | Chọn chế độ | 89H | Không |
| TL0 | Byte thấp của bộ định thời 0 | 8AH | Không |
| TL1 | Byte thấp của bộ định thời 1 | 8BH | Không |
| TH0 | Byte cao của bộ định thời 0 | 8CH | Không |
| TH1 | Byte cao của bộ định thời 1 | 8DH | Không |
| T2CON | Điều khiển bộ định thời 2 | C8H | Có |
| RCAP2L | Nhận byte thấp của bộ định thời 2 | CAH | Không |
| RCAP2H | Nhận byte cao của bộ định thời 2 | CBH | Không |
| TL2 | Byte thấp của bộ định thời 2 | CCH | Không |
| TH2 | Byte cao của bộ định thời 2 | CDH | Không |

Bảng 4.1 : Các thanh ghi chức năng đặc biệt của bộ định thời

## 4.2 THANH GHI CHẾ ĐỘ ĐỊNH THỜI ( TMOD )

Thanh ghi TMOD ( timer mode register ) chứa hai nhóm 4-bit dùng để thiết lập chế độ hoạt động cho bộ định thời 0 và bộ định thời 1 ( xem bảng 4.2 và 4.3 ). TMOD không được định địa chỉ từng bit và điều này cũng không cần thiết. Một cách tổng quát, TMOD được nạp một lần bởi

phần mềm ở thời điểm bắt đầu của một chương trình để khởi động chế độ hoạt động của bộ định thời. Sau đó, bộ định thời có thể được dừng, bắt đầu, v.v... bằng cách truy xuất các thanh ghi chức năng đặc biệt khác của bộ định thời.

| ***Bit*** | ***Tên*** | ***Bộ định thời*** | ***Mô tả*** |
|---|---|---|---|
| 7 | GATE | 1 | Bit điều khiển cổng. Khi được *set* lên 1, bộ định thời chỉ hoạt động trong khi $\overline{INT1}$ ở mức cao. |
| 6 | C/$\overline{T}$ | 1 | Bit chọn chức năng đếm hoặc định thời : 1 = đếm sự kiện 0 = định thời trong một khoảng thời gian. |
| 5 | M1 | 1 | Bit chọn chế độ thứ nhất (xem bảng 4.3) |
| 4 | M0 | 1 | Bit chọn chế độ thứ hai (xem bảng 4.3) |
| 3 | GATE | 0 | Bit điều khiển cổng cho bộ định thời 0 |
| 2 | C/$\overline{T}$ | 0 | Bit chọn chức năng đếm hoặc định thời cho bộ định thời 0 |
| 1 | M1 | 0 | Bit chọn chế độ thứ nhất |
| 0 | M0 | 0 | Bit chọn chế độ thứ hai. |

Bảng 4.2 : Thanh ghi chọn chế độ định thời

| ***M1*** | ***M0*** | ***Chế độ*** | ***Mô tả*** |
|---|---|---|---|
| 0 | 0 | 0 | Chế độ định thời 13-bit |
| 0 | 1 | 1 | Chế độ định thời 16-bit |
| 1 | 0 | 2 | Chế độ tự động nạp lại 8-bit |
| 1 | 1 | 3 | Chế độ định thời chia xẻ<br>Bộ định thời 0 : TL0 là một bộ định thời 8-bit được điều khiển bởi các bit chọn chế độ của bộ định thời 0. TH0, tương tự TL0 chỉ khác là được điều khiển bởi các bit chọn chế độ của bộ định thời 1.<br>Bộ định thời 1 : dừng, không hoạt động. |

Bảng 4.3 : Các chế độ định thời

## 4.3 THANH GHI ĐIỀU KHIỂN ĐỊNH THỜI ( TCON )

Thanh ghi TCON chứa các bit điều khiển và trạng thái của bộ định thời 0 và bộ định thời 1 ( xem bảng 4.4 ). Bốn bit cao trong TCON ( TCON.4 – TCON.7 ) được dùng để điều khiển các bộ định thời hoạt động hoặc ngưng ( TR0, TR1 ) hoặc để báo các bộ định thời tràn (TF0, TF1). Các bit này được dùng rộng rãi trong các thí dụ của chương này.

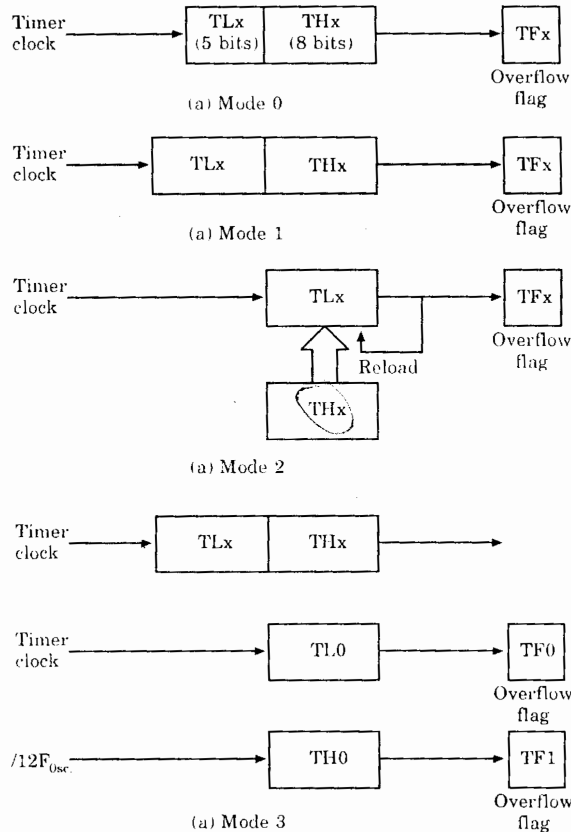
Bốn bit thấp của TCON ( TCON.0 – TCON.3 ) không dùng để điều khiển các bộ định thời, chúng được dùng để phát hiện và khởi động các ngắt ngoài. Việc thảo luận về các bit này được hoãn lại cho đến chương 6, chương đề cập đến các ngắt.

| ***Bit*** | ***Ký hiệu*** | ***Địa chỉ bit*** | ***Mô tả*** |
|---|---|---|---|
| TCON.7 | TF1 | 8FH | Cờ tràn của bộ định thời 1. Cờ này được *set* bởi phần cứng khi có tràn, được xóa bởi phần mềm, hoặc bởi phần cứng khi bộ vi xử lý trỏ đến trình phục vụ ngắt. |
| TCON.6 | TR1 | 8EH | Bit điều khiển hoạt động của bộ định thời 1. Bit này được *set* hoặc được xóa bởi phần mềm để điều khiển bộ định thời hoạt động hay ngưng hoạt động. |
| TCON.5 | TF0 | 8DH | Cờ tràn của bộ định thời 0 |
| TCON.4 | TR0 | 8CH | Bit điều khiển hoạt động của bộ định thời 0. |
| TCON.3 | IE1 | 8BH | Cờ ngắt bên ngoài 1 ( kích khởi cạnh ). Cờ này được *set* bởi phần cứng khi có cạnh âm ( xuống ) xuất hiện trên chân $\overline{INT1}$, được xóa bởi phần mềm, hoặc phần cứng khi CPU trỏ đến trình phục vụ ngắt. |
| TCON.2 | IT1 | 8AH | Cờ ngắt bên ngoài 1 ( kích khởi cạnh hoặc mức ). Cờ này được *set* hoặc xóa bởi phần mềm khi xảy ra cạnh âm ( xuống ) hoặc mức thấp tại chân ngắt ngoài. |
| TCON.1 | IE0 | 89H | Cờ ngắt bên ngoài 0 ( kích khởi cạnh ). |
| TCON.0 | IT0 | 88H | Cờ ngắt bên ngoài 0 ( kích khởi cạnh hoặc mức ). |

Bảng 4.4 : Thanh ghi điều khiển định thời TCON

## 4.4 CÁC CHẾ ĐỘ ĐỊNH THỜI VÀ CỜ TRÀN

Từng bộ định thời sẽ được đề cập đến trong mục này. Do ta có hai bộ định thời trên *chip* 8051, ký hiệu " x " được sử dụng để chỉ hoặc bộ định thời 0 hoặc bộ định thời 1; thí dụ THx có nghĩa là TH0 hoặc TH1 tùy theo bộ định thời là 0 hay 1. Sự sắp xếp các thanh ghi TLx, THx và các cờ tràn TFx của bộ định thời được trình bày trong hình 4.2 cho từng chế độ định thời.




Hình 4.2 : Các chế độ định thời (a) chế độ 0 (b) chế độ 1 (c) chế độ 2 (d) chế độ 3

Timer *clock* : xung *clock* cho bộ định thời

Overflow flag : cờ tràn

Mode 0, 1, 2. 3 : chế độ 0, 1, 2, 3

Reload : nạp lại

1/12 $F_{osc}$ : 1/12 tần số của mạch dao động trên *chip*

#### 4.4.1 Chế độ định thời 13-bit ( chế độ 0 )

Chế độ 0 là chế độ định thời 13-bit cung cấp khả năng tương thích với bộ vi điều khiển tiền nhiệm 8048. Chế độ này không được dùng cho các thiết kế mới ( xem hình 4.2a ). Byte cao của bộ định thời THx được ghép *cascade* với 5bit thấp của byte thấp của bộ định thời TLx để tạo thành một bộ định thời 13-bit. Ba bit cao của TLx không sử dụng.

#### 4.4.2 Chế độ định thời 16-bit ( chế độ 1 )

Chế độ 1 là chế độ định thời 16-bit và có cấu hình giống chế độ định thời 13-bit, chỉ khác nhau ở chỗ bây giờ là bộ định thời 16-bit. Xung *clock* đặt vào các thanh ghi định thời cao và thấp kết hợp ( TLx/THx ). Khi có xung *clock* đến, bộ định thời đếm lên : 0000H, 0001H, 0002H, .... Một tràn sẽ xuất hiện khi có sự chuyển số đếm từ FFFFH xuống 0000H, sự kiện này sẽ *set* cờ tràn bằng 1 và bộ định thời tiếp tục đếm. Cờ tràn là bit TFx trong thanh ghi điều khiển định thời TCON, bit này được đọc hoặc ghi bởi phần mềm ( xem hình 4.2.b ).

Bit có ý nghĩa lớn nhất ( MSB ) của giá trị trong các thanh ghi định thời là bit 7 của THx và bit có ý nghĩa thấp nhất ( LSB ) là bit 0 của TLx. Bit LSB thay đổi trạng thái và chia 2 tần số xung *clock* định thời ở ngõ vào trong khi bit MSB thay đổi trạng thái và chia cho 65536 ( tức là $2^{16}$ ) tần số xung *clock* định thời ở ngõ vào. Các thanh ghi định thời ( TLx / THx ) có thể được đọc hoặc ghi ở một thời điểm bất kỳ bởi phần mềm.

#### 4.4.3 Chế độ tự nạp lại 8-bit ( chế độ 2 )

Chế độ 2 là chế độ tự nạp lại 8-bit. Byte thấp của bộ định thời ( TLx ) hoạt động định thời 8-bit trong khi byte cao của bộ định thời lưu giữ giá trị nạp lại. Khi số đếm tràn từ FFH xuống 00H, không chỉ cờ tràn của bộ định thời được *set* lên 1 mà giá trị trong THx còn được nạp vào TLx ; việc đếm sẽ tiếp tục từ giá trị này cho đến khi xảy ra 1 tràn ( FFH → 00H ) kế tiếp, v.v... Chế độ này khá tiện lợi do bởi việc tràn bộ định thời xảy ra ở những khoảng thời gian xác định và tuần hoàn một khi các thanh ghi TMOD và THx đã được khởi động ( xem hình 4.2c ).

#### 4.4.4 Chế độ định thời chia xẻ ( chế độ 3 )

Chế độ 3 là chế độ định thời chia xẻ và có hoạt động khác nhau cho từng bộ định thời. Bộ định thời 0 ở chế độ 3 được chia thành 2 bộ định thời 8-bit hoạt động riêng rẽ TL0 và TH0, mỗi bộ định thời sẽ *set* các cờ tràn tương ứng TF0 và TF1 khi xảy ra tràn.

Bộ định thời 1 không hoạt động ở chế độ 3 nhưng có thể được khởi động bằng cách chuyển bộ định thời này vào một trong các chế độ khác.

Giới hạn duy nhất là cờ tràn TF1 của bộ định thời 1 không bị ảnh hưởng bởi bộ định thời 1 khi bộ này xảy ra tràn vì TF1 được nối với bộ định thời 8-bit TH0.

Chế độ 3 chủ yếu cung cấp thêm một bộ định thời 8 bit nữa ; nghĩa là 8051 có thêm bộ định thời thứ ba. Khi bộ định thời 0 ở chế độ 3, bộ định thời 1 có thể hoạt động hoặc ngưng bằng cách chuyển bộ này ra khỏi chế độ 3 hoặc vào chế độ 3. Bộ định thời 1 có thể được sử dụng bởi *port* nối tiếp ( lúc này bộ định thời 1 làm nhiệm vụ của bộ tạo xung *clock* tốc độ baud ) hoặc được sử dụng theo một cách nào đó nhưng không yêu cầu ngắt ( vì bộ định thời 1 lúc này không còn nối với TF1 ).

## 4.5 NGUỒN XUNG *CLOCK* ĐỊNH THỜI

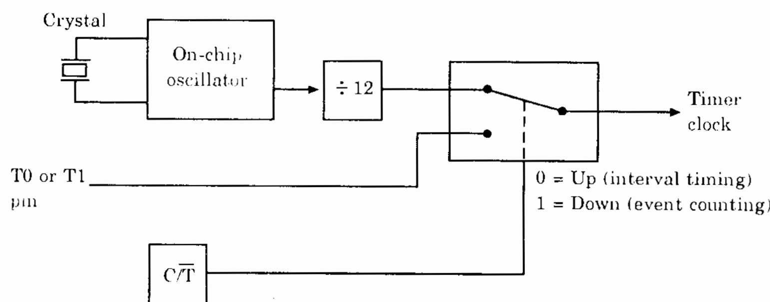
Hình 4.2 không trình bày cách tạo ra nguồn xung *clock* định thời cho các bộ định thời. Có 2 khả năng tạo ra nguồn xung *clock* này, việc lựa chọn khả năng nào do ta thiết lập bit C/$\overline{T}$ ( counter / timer ) của thanh ghi TMOD bằng 1 hay 0 khi bộ định thời được khởi động. Một nguồn xung *clock* được dùng để định thời trong một khoảng thời gian, nguồn xung *clock* còn lại được dùng để đếm sự kiện.

### 4.5.1 Định thời một khoảng thời gian

Nếu C/$\overline{T}$ = 0, hoạt động định thời được chọn và nguồn xung *clock* của bộ định thời do mạch dao động bên trong *chip* tạo ra. Một mạch chia 12 tầng được thêm vào để giảm tần số xung *clock* đến một giá trị thích hợp với hầu hết các ứng dụng. Lúc này bộ định thời được dùng để định thời trong một khoảng thời gian. Các thanh ghi định thời ( TLx / THx ) đếm lên với tần số xung *clock* bằng 1/12 tần số của mạch dao động trên *chip* [ nghĩa là nếu thạch anh là 12MHz, tần số xung *clock* là 1MHz ]. Bộ định thời sẽ tràn sau một số xung *clock* cố định phụ thuộc vào giá trị ban đầu nạp cho các thanh ghi định thời ( TLx / THx ).

### 4.5.2 Đếm sự kiện

Nếu C/$\overline{T}$ = 1, bộ định thời được cung cấp xung *clock* từ 1 nguồn tạo xung bên ngoài. Trong đa số các ứng dụng, nguồn xung *clock* này cung cấp cho bộ định thời một xung dựa trên việc xảy ra một sự kiện – bộ định thời bây giờ đếm sự kiện. Số các sự kiện được xác định trong phần mềm bằng cách đọc các thanh ghi định thời ( TLx / THx ), giá trị 16-bit trong các thanh ghi này tăng theo mỗi sự kiện. Hai chân của *port* 3 ( P3.4 và P3.5 ) bây giờ trở thành ngõ vào xung *clock* cho các bộ định thời. Chân P3.4 là ngõ vào xung *clock* cho bộ định thời 0 ( ta còn gọi là chân T0 ở ngữ cảnh này ), chân P3.5 hoặc T1 là ngõ vào xung *clock* cho bộ định thời 1 ( xem hình 4.3 ).




Hình 4.3 : Nguồn xung *clock*

Crystal : tinh thể thạch anh

On-chip oscillator : bộ dao động bên trong *chip*

T0 or T1 pin : chân T0 hoặc T1

0 = Up ( interval timing ) : 0 = vị trí trên ( định thời một khoảng thời gian )

1 = Down ( event counting ) : 1 = vị trí dưới ( đếm sự kiện )

Timer clock : xung *clock* định thời

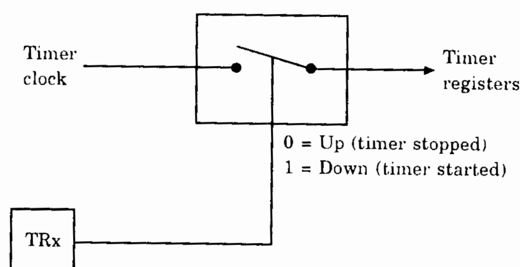
Trong các ứng dụng đếm sự kiện, các thanh ghi định thời tăng mỗi khi xảy ra chuyển trạng thái từ 1 xuống 0 ở ngõ vào Tx ( T0 hoặc T1 ). Ngõ vào Tx được lấy mẫu trong suốt thời gian S5P2 của mỗi một chu kỳ máy, vậy thì khi ngõ vào ở mức cao trong một chu kỳ và mức thấp trong chu kỳ kế, số đếm được tăng. Giá trị mới xuất hiện trong các thanh ghi định thời trong suốt thời gian S3P1 của chu kỳ tiếp theo chu kỳ phát hiện sự chuyển trạng thái. Từ đó ta thấy phải mất 2 chu kỳ máy ( 2µs ) để nhận biết sự chuyển trạng thái từ 1 xuống 0, tần số cực đại của nguồn xung *clock* bên ngoài là 500 KHz ( với giả sử *chip* vi điều khiển hoạt động với thạch anh 12 MHz ).

## 4.6 KHỞI ĐỘNG, DỪNG VÀ ĐIỀU KHIỂN CÁC BỘ ĐỊNH THỜI

Hình 4.2 minh họa các cấu hình khác nhau của các thanh ghi định thời TLx, THx và các cờ tràn của bộ định thời, TFx. Hai khả năng cung cấp xung *clock* cho bộ định thời cho ở hình 4.3. Bây giờ chúng ta khảo sát việc khởi động, dừng và điều khiển các bộ định thời.

Cách đơn giản nhất để khởi động và dừng các bộ định thời là sử dụng bit điều khiển hoạt động TRx trong thanh ghi TCON. TRx được xóa khi thiết lập lại hệ thống ; nghĩa là các bộ định thời ngưng hoạt động. ( xem hình 4.4 ).

Do thanh ghi TCON là thanh ghi được định địa chỉ từng bit, ta dễ dàng khởi động hoặc dừng các bộ định thời bằng chương trình.




Hình 4.4 : Bắt đầu và dừng các bộ định thời

Timer clock : xung *clock* định thời

Timer registers : các thanh ghi định thời

0 = Up ( timer stopped ) : 0 = vị trí trên ( bộ định thời dừng )

1 = Down ( timer started ) : 1 = vị trí dưới ( bộ định thời được khởi động )

Thí dụ bộ định thời 0 được khởi động bằng lệnh :

```
SETB    TR0
```

và được điều khiển dừng bằng lệnh :

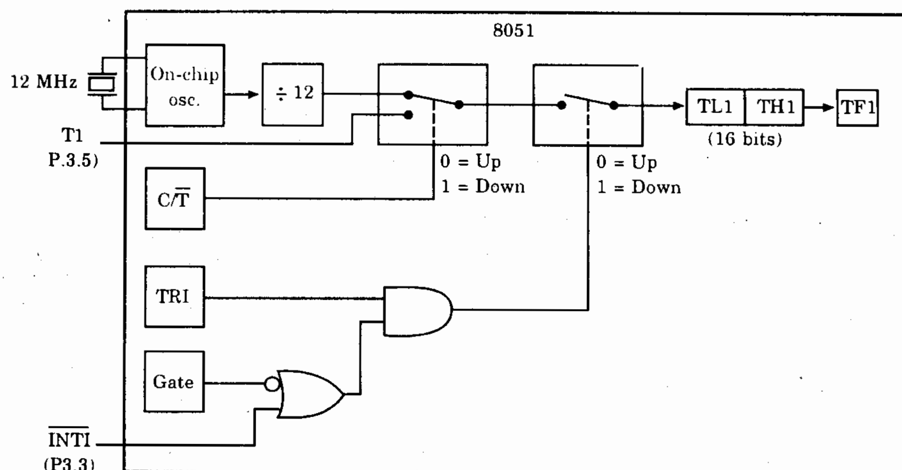
```
CLR     TR0
```

Trình dịch hợp ngữ sẽ thực hiện việc biến đổi ký hiệu " TR0 " thành địa chỉ bit. Lệnh SETB TR0 đồng nghĩa với lệnh SETB 8CH.

Một phương pháp khác để điều khiển các bộ định thời là sử dụng bit GATE trong thanh ghi TMOD và ngõ vào $\overline{INTx}$. Bằng cách *set* bit GATE lên 1 ta cho phép bộ định thời được điều khiển bởi $\overline{INTx}$. Phương pháp này thường được dùng để đo độ rộng xung như được trình bày sau đây.

Giả sử $\overline{INT0}$ ở mức thấp rồi chuyển sang mức cao trong một khoảng thời gian và ta đo khoảng thời gian này. Ta khởi động bộ định thời 0 ở chế độ 2, chế độ định thời 16-bit với TL0/TH0 = 0000H, GATE = 1 và TR0 = 1. Khi $\overline{INT0}$ chuyển lên mức cao, bộ định thời được mở cổng và nhận xung *clock* có tần số 1MHz. Khi $\overline{INT0}$ chuyển xuống mức thấp, bộ định thời bị khóa cổng không nhận xung *clock* nữa và độ rộng xung tính bằng μs là số đếm trong TL0 / TH0 ( $\overline{INT0}$ có thể được lập trình để tạo ra 1 ngắt khi chân này chuyển trở về mức thấp ).

Hình 4.5 minh họa bộ định thời 1 hoạt động ở chế độ 1 ( bộ định thời 16-bit ). Cùng với các thanh ghi TL1/TH1 và cờ tràn TF1, sơ đồ ở hình 4.5 trình bày các khả năng cấp nguồn xung *clock*, khởi động, dừng và điều khiển bộ định thời 1.




Hình 4.5 : Hoạt động ở chế độ 1 của bộ định thời 1

On-chip osc : mạch dao động bên trong *chip*

## 4.7 KHỞI ĐỘNG VÀ TRUY XUẤT CÁC THANH GHI ĐỊNH THỜI

Các bộ định thời thường được khởi động một lần ở thời điểm bắt đầu chương trình để thiết lập chế độ hoạt động theo yêu cầu. Trong thân của chương trình, các bộ định thời được điều khiển hoạt động, dừng, kiểm tra các bit cờ và xóa, các thanh ghi định thời được đọc hoặc cập nhật và v.v... tùy theo yêu cầu của ứng dụng.

TMOD là thanh ghi được khởi động trước tiên vì đây là thanh ghi thiết lập chế độ hoạt động. Lấy thí dụ lệnh sau đây khởi động bộ định thời 1 hoạt động ở chế độ 16-bit ( chế độ 1 ), xung *clock* được cấp từ mạch dao động trên *chip* ( định thời một khoảng thời gian ) :

```
MOV   TMOD, #00010000B
```

Kết quả của lệnh này là thiết lập M1 = 0 và M0 = 1 để ấn định chế độ 1, C/$\overline{T}$ = 0 và GATE = 0 để sử dụng xung *clock* trên *chip*, xóa các bit chọn chế độ của bộ định thời 0 ( xem bảng 4.2 ). Dĩ nhiên trên thực tế bộ định thời không bắt đầu công việc định thời cho đến khi bit điều khiển hoạt động TR1 được *set* bằng 1.

Trong trường hợp cần đến số đếm ban đầu, các thanh ghi định thời TL1/TH1 cũng phải được khởi động. Cần nhớ là các bộ định thời đếm lên và thiết lập cờ tràn bằng 1 khi xảy ra trận số đếm từ FFFFH xuống 0000H, vậy thì một khoảng thời gian 100 μs có thể được định thời bằng cách khởi động TL1/TH1 chứa số đếm nhỏ hơn 0000H một lượng là 100, nghĩa là –100 hay FF9CH. Các lệnh sau thực hiện điều này :

```
MOV  TL1, #9CH
MOV  TH1, #0FFH
```

Kế đến bộ định thời bắt đầu hoạt động bằng cách thiết lập bit điều khiển hoạt động bằng 1 như sau :

```
SETB  TR1
```

Cờ tràn được tự động thiết lập sau khoảng thời gian 100 µs. Phần mềm có thể chứa một vòng lặp trì hoãn thời gian 100 µs bằng cách sử dụng một lệnh rẽ nhánh và lặp lại chính lệnh này trong khi cờ tràn chưa được *set* bằng 1 :

```
WAIT:  JNB  TF1, WAIT
```

Khi bộ định thời tràn, ta cần dừng bộ định thời và xóa cờ tràn bằng phần mềm :

```
CLR   TR1        ; dừng bộ định thời 1
CLR   TF1        ; xóa cờ tràn
```

**Đọc bộ định thời đang hoạt động**

Trong một số ứng dụng ta cần phải đọc giá trị ( nội dung ) chứa trong các thanh ghi định thời đang hoạt động.

Do ta phải đọc 2 thanh ghi định thời bằng 2 dòng lệnh liên tiếp ( do không có lệnh đọc đồng thời cả hai thanh ghi định thời này ), một sai pha ( phase error ) có thể xuất hiện nếu có tràn từ byte thấp chuyển sang byte cao giữa 2 lần đọc và do vậy ta không thể đọc đúng được giá trị cần đọc. Giải pháp đưa ra là trước tiên ta phải đọc byte cao, kế đến đọc byte thấp và rồi đọc byte cao lần nữa. Nếu byte cao thay đổi giá trị, ta lặp lại các thao tác đọc vừa nêu. Các lệnh sau đây đọc nội dung của các thanh ghi định thời TL1 / TH1, đưa vào các thanh ghi R6 / R7 và giải quyết vấn đề vừa nêu :

```
AGAIN :     MOV    A, TH1
            MOV    R6, TL1
            CJNE   A, TH1, AGAIN
            MOV    R7, A
```

## 4.8 KHOẢNG THỜI GIAN NGẮN VÀ KHOẢNG THỜI GIAN DÀI

Tầm nào của các khoảng thời gian có thể định thời được ? Vấn đề này được khảo sát bằng cách giả sử 8051 hoạt động với thạch anh nối với mạch dao động nội có tần số hoạt động là 12 MHz. Mạch dao động trên *chip* được chia cho 12 và các xung *clock* cấp cho mạch định thời có

tần số 1 MHz. Khoảng thời gian ngắn nhất có thể định thời được bị giới hạn không phải bởi tần số của xung *clock* định thời mà bởi phần mềm nghĩa là thời gian thực thi các lệnh tạo ra giới hạn đối với các khoảng thời gian định thời rất ngắn. Lệnh ngắn nhất của 8051 thực hiện trong một chu kỳ máy hay 1 µs. Bảng 4.5 tóm tắt các kỹ thuật được dùng để tạo ra các khoảng thời gian định thời khác nhau.

| ***Khoảng thời gian*** | ***Kỹ thuật*** |
|---|---|
| ≈ 10 | Điều chỉnh phần mềm |
| 256 | Bộ định thời 8-bit tự động nạp lại. |
| 65536 | Bộ định thời 16-bit |
| không giới hạn | Bộ định thời 16 bit + các vòng lặp |

Bảng 4.5 : Khoảng thời gian định thời cực đại ( µs )

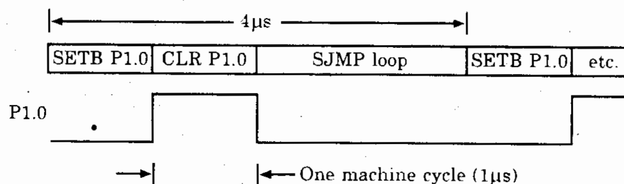
**Thí dụ 4.1** : Tạo dạng xung

*Viết 1 chương trình tạo dạng xung tuần hoàn trên chân P1.0 có tần số cao có thể có được. Tần số và chu kỳ nhiệm vụ của dạng xung là bao nhiêu ?*

Các khoảng thời gian rất ngắn ( có nghĩa là tần số cao ) có thể được lập trình mà không cần sử dụng đến các bộ định thời. Thí dụ chương trình sau :

```
        ORG   8100H
LOOP:   SETB  P1.0        ; 1 chu kỳ máy
        CLR   P1.0        ; 1 chu kỳ máy
        SJMP  LOOP        ; 2 chu kỳ máy
        END
```

Chương trình trên tạo ra dạng xung trên chân P1.0, xung có chu kỳ là 4 µs : thời gian mức cao là 1 µs và thời gian mức thấp là 3 µs trong một chu kỳ. Tần số của xung là 250 KHz và chu kỳ nhiệm vụ là 25% (xem hình 4.6)




Hình 4.6 : Dạng xung của thí dụ 4.1

SJMP loop : vòng lặp SJMP
One machine cycle ( 1 μs ) : một chu kỳ máy ( 1μs )

Lệnh SETB P1.0 thực tế không *sét* bit 0 của *port* 1 lên 1 cho đến khi kết thúc lệnh này, trong thời gian S6P2, và các lệnh tiếp theo cũng tương tự. Chu kỳ của tín hiệu ngõ ra có thể kéo dài thêm một ít bằng cách chèn các lệnh NOP ( không toán hạng ) vào trong vòng lặp. Mỗi lệnh NOP được chèn thêm làm cho chu kỳ của tín hiệu ngõ ra được cộng thêm 1 μs. Thí dụ nếu ta chèn 2 lệnh NOP sau lệnh SETB P1.0, chương trình sẽ tạo ở ngõ ra một xung vuông có chu kỳ 6 μs và tần số là 166.7 KHz. Tuy vậy ta cần thấy rằng việc điều chỉnh phần mềm như trên sẽ trở nên cồng kềnh và vụng về, cách lựa chọn tốt nhất để trì hoãn vẫn là sử dụng bộ định thời.

Các khoảng thời gian dài vừa phải dễ dàng nhận được bằng cách sử dụng chế độ tự nạp lại 8-bit, chế độ 2. Do các khoảng thời gian cần được định thời được thiết lập bởi một số đếm 8-bit, khoảng thời gian dài nhất có thể có trước khi tràn là $2^8 = 256$ μs.

**Thí dụ 4.2** : Sóng vuông 10 KHz

*Viết 1 chương trình tạo sóng vuông 10 KHz trên chân P1.0 bằng cách sử dụng bộ định thời 0.*

Sóng vuông 10 KHz yêu cầu chu kỳ 100 μs với thời gian mức cao là 50 μs và thời gian mức thấp là 50 μs. Do khoảng thời gian này nhỏ hơn 256 μs nên chế độ 2 được sử dụng. Một tràn xảy ra sau mỗi 50 μs yêu cầu một giá trị số đếm nhỏ hơn 00H một lượng là +50 phải được nạp và nạp lại cho TL0, nghĩa là giá trị nạp cho TH0 là –50. Dưới đây là chương trình theo yêu cầu :

```
        ORG     8100H
        MOV     TMOD, #02H          ; chế độ tự nạp lại 8-bit
        MOV     TH0, #-50H          ; TH0 chứa giá trị -50
        SETB    TR0                 ; bộ định thời hoạt động
LOOP:   JNB     TF0, LOOP           ; chờ tràn
        CLR     TF0                 ; xóa cờ tràn
        CPL     P1.0                ; đổi trạng thái bit P1.0
        SJMP    LOOP                ; lặp lại
        END
```

Chương trình trên sử dụng lệnh lấy bù bit CPL thay vì là lệnh SETB và CLR như trong thí dụ 4.1. Giữa hai thao tác lấy bù có một trì hoãn

| *Bit* | *Ký hiệu* | *Địa chỉ bit* | *Mô tả* |
|---|---|---|---|
| T2CON.7 | TF2 | CFH | Cờ tràn của bộ định thời 2. (không được *set* khi TCLK hoặc RCLK = 1) |
| T2CON.6 | EXF2 | CEH | Cờ ngoài của bộ định thời 2. Cờ được *set* khi có sự thu nhận hoặc nạp lại xảy ra do bởi sự chuyển trạng thái 1 → 0 ở chân T2EX và EXEN2 = 1 ; khi các ngắt do bộ định thời được phép, EXF2 = 1 làm cho CPU trở tới trình phục vụ ngắt. EXF2 được xóa bởi phần mềm. |
| T2CON.5 | RCLK | CDH | *Clock* thu của bộ định thời 2. Khi được *set*, bộ định thời 2 cung cấp tốc độ baud (khi thu) cho *port* nối tiếp; bộ định thời 1 cung cấp tốc độ baud (khi phát). |
| T2CON.4 | TCLK | CCH | *Clock* phát của bộ định thời 2. Khi được *set*, bộ định thời 2 cung cấp tốc độ baud phát; bộ định thời 1 cung cấp tốc độ baud thu. |
| T2CON.3 | EXEN2 | CBH | Cho phép từ bên ngoài. Khi được *set*, việc thu nhận và nạp lại xuất hiện khi có sự chuyển trạng thái 1 → 0 ở chân T2EX. |
| T2CON.2 | TR2 | CAH | Bit điều khiển hoạt động bộ định thời 2. Bit này được *set* hay xóa bởi phần mềm để điều khiển bộ định thời 2 hoạt động hoặc ngưng. |
| T2CON.1 | $C/\overline{T}2$ | C9H | Bit chọn chức năng đếm hoặc định thời của bộ định thời 2. 1 = đếm sự kiện; 0 = định thời một khoảng thời gian. |
| T2CON.0 | $CP/\overline{RL2C}$ | C8H | Cờ thu nhận/nạp lại của bộ định thời 2. Khi được *set*, việc thu nhận xuất hiện khi có sự chuyển trạng thái 1 → 0 ở T2EX nếu EXEN2 = 1; khi được xoá, tự động nạp lại xuất hiện khi có tràn bộ định thời hoặc sự chuyển trạng thái ở T2EX nếu EXEN2 = 1 ; nếu RCLK hoặc TCLK = 1, bit này được bỏ qua. |

Bảng 4.6 : Thanh ghi điều khiển định thời T2CON

Việc nạp lại xảy ra khi có tràn số đếm từ FFFFH xuống 0000H ở TL2/TH2 và thiết lập cờ TF2 bằng 1. Điều kiện này được xác định bởi phần mềm hoặc được lập trình để tạo ra một ngắt. Với cả 2 cách vừa nêu, TF2 phải được xóa bởi phần mềm trước khi cờ này được *set* lần nữa.

Bằng cách *set* bit EXEN2 trong thanh ghi T2CON bằng 1, việc nạp lại cũng xuất hiện khi có sự chuyển trạng thái 1 → 0 của tín hiệu đặt vào chân T2EX ( chân P1.1 ). Sự chuyển trạng thái 1 → 0 ở T2EX cũng thiết lập bằng 1 cho một bit cờ mới trong bộ định thời 2 : bit EXF2. Cùng với TF2, bit EXF2 cũng được kiểm tra bởi phần mềm hoặc tạo ra một ngắt. EXF2 phải được xóa bởi phần mềm. Chế độ tự nạp lại của bộ định thời 2 được trình bày ở hình 4.9.

### 4.9.2 Chế độ thu nhận

Khi CP/$\overline{RL2}$ = 1, chế độ thu nhận được chọn. Bộ định thời 2 hoạt động như một bộ định thời 16-bit và thiết lập cờ TF2 bằng 1 khi xảy ra tràn số đếm từ FFFFH xuống 0000H ở TL2/TH2. Trạng thái của TF2 được kiểm tra bởi phần mềm hoặc tạo ra 1 ngắt.

Để cho phép chế độ này hoạt động, bit EXEN2 trong T2CON phải được *set* bằng 1. Nếu EXEN2 = 1, sự chuyển trạng thái 1 → 0 ở T2EX ( P1.1 ) sẽ đưa giá trị trong các thanh ghi định thời TL2 / TH2 vào trong các thanh ghi RCAP2L và RCAP2H. Việc nạp giá trị này được điều khiển bởi xung *clock*. Cờ EXF2 trong T2CON cũng được *set* và như đã đề cập ở phần trên, được kiểm tra bởi phần mềm hoặc tạo ra một ngắt.

Chế độ thu nhận của bộ định thời 2 được trình bày ở hình 4.10.

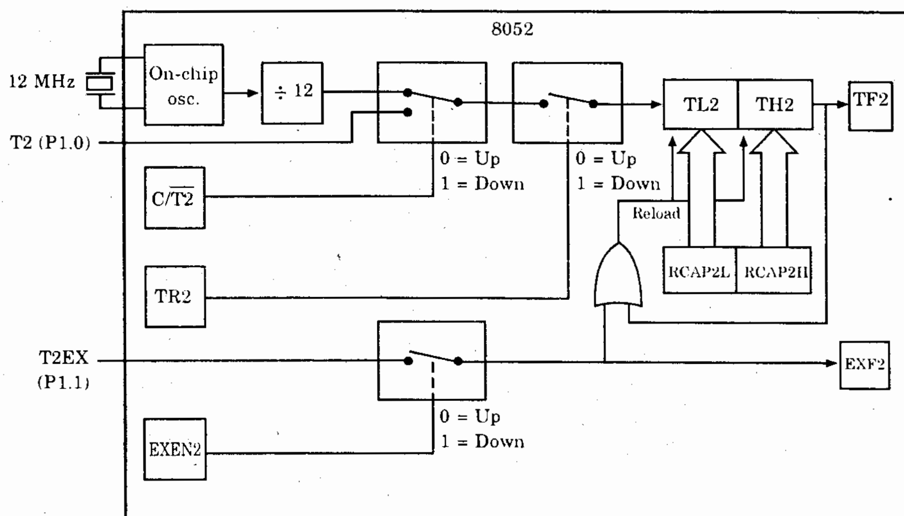
## 4.10 TẠO TỐC ĐỘ BAUD

Một ứng dụng khác của bộ định thời là cung cấp xung *clock* tốc độ baud cho *port* nối tiếp của *chip* vi điều khiển. Bộ định thời 1 ở 8051 hoặc bộ định thời 1, bộ định thời 2 ở 8052 thực hiện được công việc này. Tạo tốc độ baud sẽ đề cập đến trong chương 5.

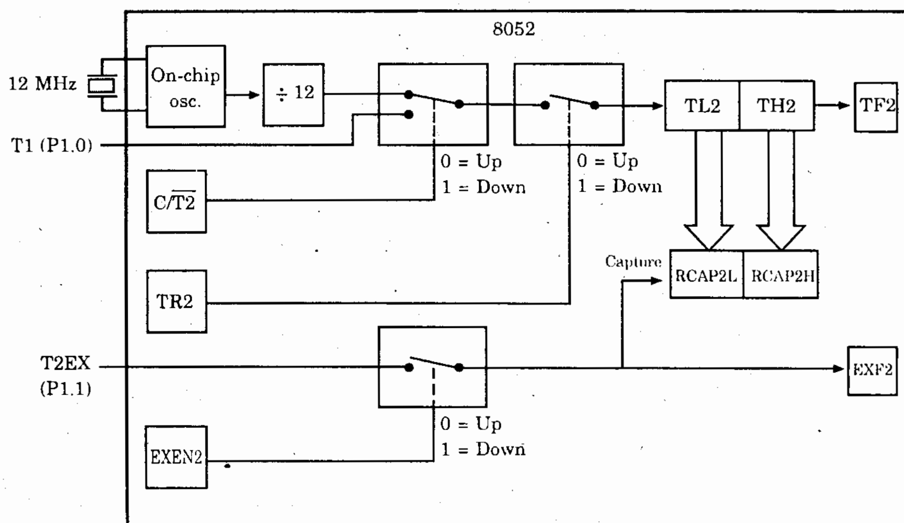
Trong chương này ta đã khảo sát các bộ định thời của các bộ vi điều khiển 8051 và 8052. Các giải pháp phần mềm cho các thí dụ làm nổi bật nét chung nhưng bị giới hạn. Các giải pháp này làm mất nhiều thời gian của CPU. Các chương trình thực thi các vòng lặp để chờ bộ định thời tràn. Điều này tốt cho các mục đích nghiên cứu học hỏi nhưng đối với các ứng dụng hướng điều khiển thực tế sử dụng các bộ vi điều khiển, CPU phải thực hiện các nhiệm vụ khác cũng như phải đáp ứng với các sự kiện bên ngoài ( như là người điều khiển đưa vào một tham số từ bàn phím chẳng hạn ). Trong chương khảo sát hoạt động ngắt, ta sẽ minh

họa việc sử dụng các bộ định thời trong môi trường được điều khiển ngắt.

Ta không cần phải kiểm tra cờ tràn của bộ định thời bằng một vòng lặp mà sẽ tạo ra một ngắt khi có tràn bộ định thời.




Hình 4.9 : Chế độ tự nạp lại 16-bit của bộ định thời 2




Hình 4.10 : Chế độ thu nhận 16-bit của bộ định thời 2

On-chip osc : bộ dao động bên trong *chip*

Reload : nạp lại

Capture : thu nhận

0 = Up : nếu bit điều khiển là 0, chuyển mạch ở vị trí trên

1 = Down : nếu bit điều khiển là 1, chuyển mạch ở vị trí dưới

# 5

# HOẠT ĐỘNG CỦA *PORT* NỐI TIẾP

## 5.1 MỞ ĐẦU

Bên trong *chip* 8051 có một *port* nối tiếp hoạt động ở một vài chế độ trên một tầm tần số rộng. Chức năng cơ bản của *port* nối tiếp là thực hiện việc chuyển đổi dữ liệu song song thành nối tiếp khi phát và chuyển đổi dữ liệu nối tiếp thành song song khi thu.

Các mạch phần cứng bên ngoài truy xuất *port* nối tiếp thông qua các chân TxD ( phát dữ liệu ) và RxD ( thu dữ liệu ) đã được giới thiệu ở chương 2. Các chân này đa hợp với hai chân của *port* 3 : P3.1 ( TxD ) và P3.0 ( RxD ).

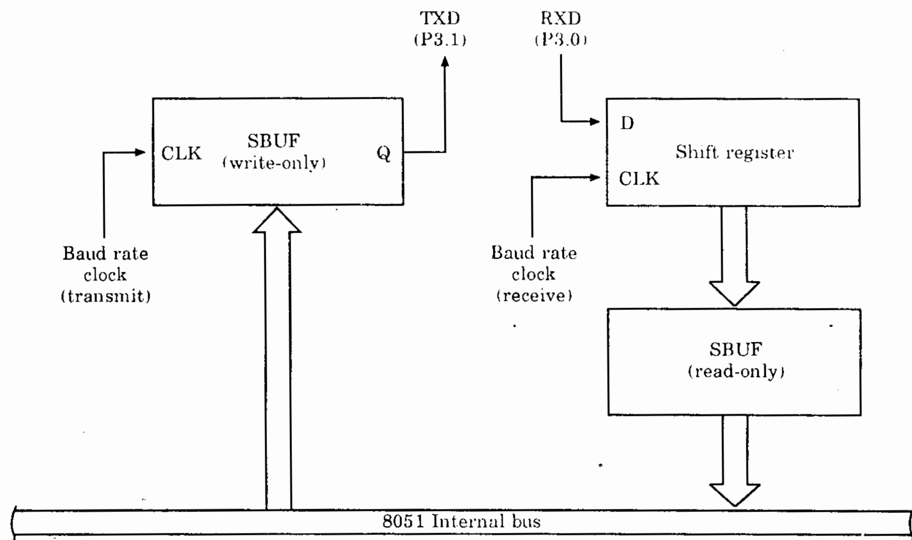
Đặc trưng của *port* nối tiếp là hoạt động song công ( full duplex ), nghĩa là có khả năng thu và phát đồng thời. Ngoài ra *port* nối tiếp còn có một đặc trưng khác, việc đệm dữ liệu khi thu của *port* này cho phép một ký tự được nhận và lưu giữ trong bộ đệm thu trong khi ký tự tiếp theo được nhận vào. Nếu CPU đọc ký tự thứ nhất trước khi ký tự thứ hai được nhận đầy đủ, dữ liệu sẽ không bị mất.

Phần mềm sử dụng hai thanh ghi chức năng đặc biệt SBUF và SCON để truy xuất *port* nối tiếp. Bộ đệm của *port* nối tiếp SBUF có địa chỉ byte là 99H, trên thực tế bao gồm hai bộ đệm. Việc ghi lên SBUF sẽ nạp dữ liệu để phát và việc đọc SBUF sẽ truy xuất dữ liệu đã nhận được. Điều này có nghĩa là ta có hai thanh ghi riêng rẽ và phân biệt : thanh ghi phát ( chỉ ghi ) hay bộ đệm phát và thanh ghi thu ( chỉ đọc ) hay bộ đệm thu ( xem hình 5.1 ).

Thanh ghi điều khiển *port* nối tiếp SCON ( có địa chỉ byte là 98H ) là thanh ghi được định địa chỉ từng bit, thanh ghi này chứa các bit trạng thái và các bit điều khiển. Các bit điều khiển sẽ thiết lập chế độ hoạt động cho *port* nối tiếp còn các bit trạng thái chỉ ra sự kết thúc việc thu

hoặc phát một ký tự. Các bit trạng thái được kiểm tra bởi phần mềm hoặc được lập trình để tạo ra ngắt.

Tần số hoạt động của *port* nối tiếp, hay còn gọi là tốc độ baud ( baud rate ) có thể cố định hoặc thay đổi. Khi tốc độ baud thay đổi được sử dụng, bộ định thời 1 cung cấp xung *clock* tốc độ baud và ta phải lập trình sao cho phù hợp. Trên 8032/8052, bộ định thời 2 cũng có thể được lập trình để cung cấp xung *clock* tốc độ baud.




Hình 5.1 : Sơ đồ khối của *port* nối tiếp

Write only : chỉ ghi

Read only : chỉ đọc

Shift register : thanh ghi dịch bit

Baud rate clock ( transmit ) : xung *clock* tốc độ baud ( phát )

Baud rate clock ( receive ) : xung *clock* tốc độ baud ( thu )

8051 internal bus : bus nội của 8051

## 5.2 THANH GHI ĐIỀU KHIỂN *PORT* NỐI TIẾP

Chế độ hoạt động của *port* nối tiếp được thiết lập bằng cách ghi từ điều khiển lên thanh ghi chọn chế độ SCON của *port* nối tiếp ở địa chỉ byte 99H ( xem bảng 5.1 và 5.2 ).

Trước khi sử dụng *port* nối tiếp, thanh ghi SCON phải được khởi động đúng chế độ yêu cầu. Thí dụ lệnh sau :

```
MOV  SCON, #01010010B
```

khởi động *port* nối tiếp ở chế độ 1 ( SM0/SM1 = 0/1 ), cho phép thu ( REN = 1 ) và *set* cờ ngắt phát bằng 1 ( TI = 1 ) để chỉ ra rằng *port* nối tiếp sẵn sàng phát dữ liệu.

## 5.3 CÁC CHẾ ĐỘ HOẠT ĐỘNG

*Port* nối tiếp của 8051 có 4 chế độ hoạt động, các chế độ được chọn bằng cách ghi 1 hoặc 0 cho các bit SM0 và SM1 trong thanh ghi SCON. Ba trong số các chế độ hoạt động cho phép truyền không đồng bộ ( asynchronous ), trong đó mỗi một ký tự được thu hoặc được phát sẽ cùng với một bit start và một bit stop tạo thành một khung ( frame ).

Nếu đã làm quen với hoạt động của *port* nối tiếp theo chuẩn RS-232C trên một máy vi tính, ta sẽ tìm thấy được sự tương tự ở các chế độ này. Với chế độ thứ tư, *port* nối tiếp hoạt động như một thanh ghi dịch bit đơn giản. Mỗi một chế độ sẽ được đề cập tóm tắt sau đây.

| Bit | Ký hiệu | Địa chỉ | Mô tả |
|---|---|---|---|
| SCON.7 | SM0 | 9FH | bit 0 chọn chế độ của *port* nối tiếp |
| SCON.6 | SM1 | 9EH | bit 1 chọn chế độ của *port* nối tiếp |
| SCON.5 | SM2 | 9DH | bit 2 chọn chế độ của *port* nối tiếp. Bit này cho phép truyền thông đa xử lý ở các chế độ 2 và 3; bit RI sẽ không được tích cực nếu bit thứ 9 nhận được là 0. |
| SCON.4 | REN | 9CH | cho phép thu. Bit này phải được *set* để nhận các ký tự. |
| SCON.3 | TB8 | 9BH | bit phát 8. Bit thứ 9 được phát ở các chế độ 2 và 3; được *set* và xóa bởi phần mềm |
| SCON.2 | RB8 | 9AH | bit thu 8. Bit thứ 9 nhận được. |
| SCON.1 | TI | 99H | cờ ngắt phát. Cờ này được *set* ngay khi kết thúc việc phát một ký tự ; được xóa bởi phần mềm |
| SCON.0 | RI | 98H | cờ ngắt thu. Cờ này được *set* ngay khi kết thúc việc thu một ký tự ; được xóa bởi phần mềm |

Bảng 5.1 : Tóm tắt thanh ghi SCON ( điều khiển *port* nối tiếp )

### 5.3.1 Thanh ghi dịch 8-bit ( chế độ 0 )

Chế độ 0, được chọn bằng cách ghi giá trị 0 vào các bit SM0 và SM1 trong thanh ghi SCON, đặt *port* nối tiếp vào chế độ thanh ghi dịch 8-bit. Dữ liệu nối tiếp được thu và phát thông qua chân RxD, chân TxD xuất xung *clock* dịch bit. Khi phát và thu dữ liệu 8-bit, bit có ý nghĩa ( giá trị vị trí ) nhỏ nhất ( bit LSB ) được thu hoặc phát trước tiên. Tốc độ baud cố định và bằng 1/12 tần số của mạch dao động trên *chip*. Các thuật ngữ " RxD " và " TxD " bị sai lệch ý nghĩa trong chế độ này. Chân RxD được sử dụng cho cả thu và phát dữ liệu còn chân TxD được dùng làm chân xuất xung *clock* dịch bit.

| SM0 | SM1 | Chế độ | Mô tả | Tốc độ baud |
|---|---|---|---|---|
| 0 | 0 | 0 | Thanh ghi dịch | Cố định ( tần số dao động / 12 ) |
| 0 | 1 | 1 | UART 8-bit | Thay đổi ( thiết lập bởi bộ định thời ) |
| 1 | 0 | 2 | UART 9-bit | Cố định ( tần số dao động / 12 hoặc / 64 ) |
| 1 | 1 | 3 | UART 9-bit | Thay đổi ( thiết lập bởi bộ định thời ) |

Bảng 5.2 : Các chế độ của *port* nối tiếp

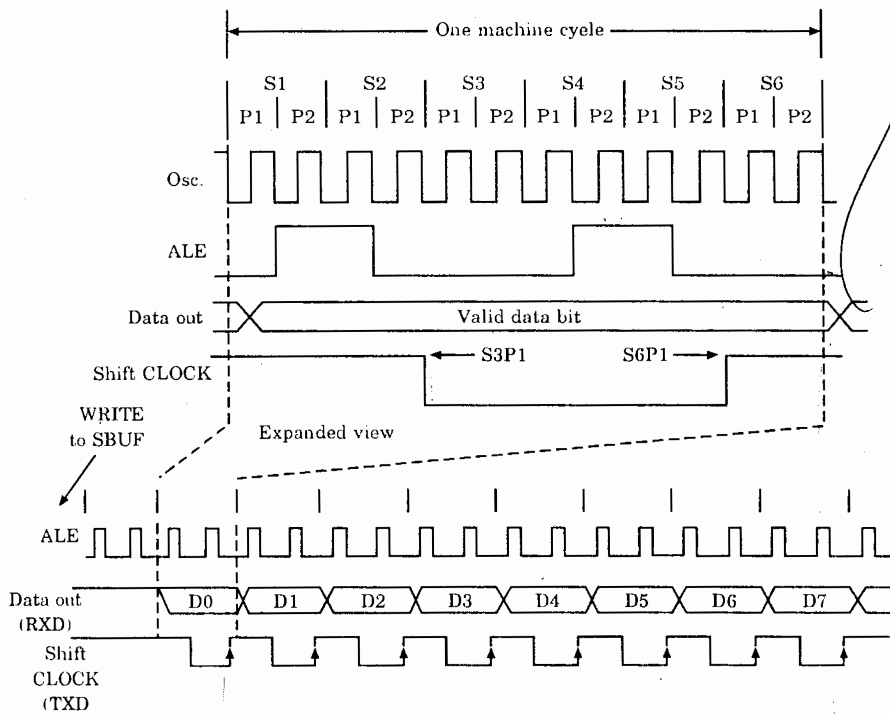
Việc phát dữ liệu được khởi động bằng một lệnh ghi dữ liệu vào SBUF. Dữ liệu được dịch ra ngoài trên chân RxD ( P3.0 ) với các xung *clock* dịch bit được gởi ra trên chân TxD ( P3.1 ). Mỗi một bit hợp lệ được truyền đi trên đường RxD trong một chu kỳ máy.

Trong mỗi một chu kỳ máy, xung *clock* dịch bit đổi thành mức thấp ở S3P1 và trở lại mức cao ở S6P1.

Giản đồ thời gian phát dữ liệu được trình bày ở hình 5.2.

Việc thu dữ liệu được khởi động khi bit cho phép thu REN ở logic 1 và cờ ngắt thu RI ở logic 0. Qui luật tổng quát là ta phải *set* bit REN bằng 1 ở thời điểm bắt đầu chương trình để khởi động *port* nối tiếp và sau đó xóa bit RI để bắt đầu công việc thu dữ liệu.

Khi bit RI được xóa, các xung *clock* dịch bit được xuất ra trên chân TxD, ta bắt đầu chu kỳ máy tiếp theo và dữ liệu được dịch vào chân RxD bởi xung *clock* dịch bit ( hiển nhiên là các mạch ghép nối để cung cấp dữ liệu trên đường RxD được đồng bộ bởi xung *clock* dịch bit trên đường TxD ( xem hình 5.3 ).




Hình 5.2 : Giản đồ thời gian phát dữ liệu ở chế độ 0

One machine cycle : một chu kỳ máy

Osc : xung *clock* của mạch dao động

ALE : xung ALE

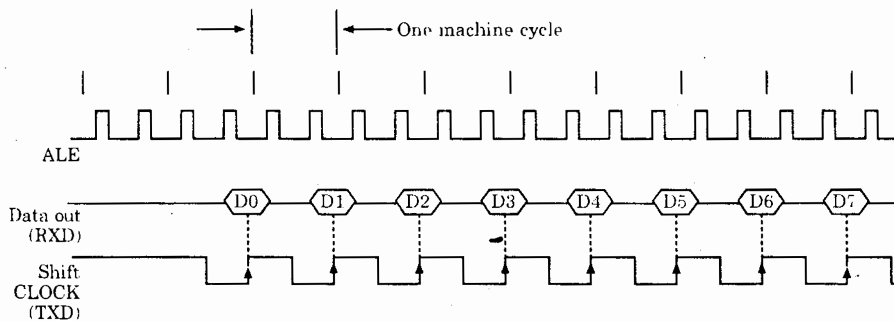
Data out : dữ liệu xuất

Valid data bit : bit dữ liệu hợp lệ

Shift CLOCK : xung *clock* dịch bit

WRITE to SBUF : ghi vào SBUF

Expanded view : quan sát được phóng đại




Hình 5.3 : Giản đồ thời gian thu dữ liệu ở chế độ 0

One machine cycle : một chu kỳ máy

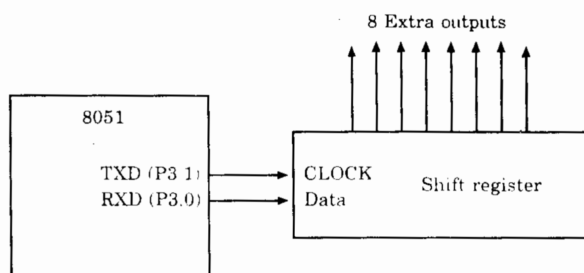
ALE : xung ALE

Data out : dữ liệu xuất

Shift CLOCK : xung *clock* dịch bit

Việc dịch dữ liệu vào port nối tiếp xảy ra ở cạnh dương ( cạnh lên ) của TxD.

Một ứng dụng khả thi của chế độ 0 ( chế độ thanh ghi dịch bit ) là mở rộng thêm các ngõ ra cho 8051. Một vi mạch thanh ghi dịch nối tiếp-song song có thể được nối với các chân TxD và RxD của 8051 để cung cấp thêm 8 đường xuất ( xem hình 5.4 ). Các thanh ghi dịch bit khác có thể ghép *cascade* với thanh ghi dịch bit đầu tiên để mở rộng thêm nữa.




Hình 5.4 : Chế độ thanh ghi dịch bit của *port* nối tiếp

Shift register : thanh ghi dịch bit

8 extra outputs : 8 ngõ ra mở rộng

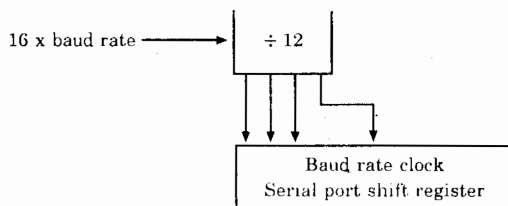
### 5.3.2 UART 8-bit có tốc độ baud thay đổi ( chế độ 1 )

Trong chế độ 1, *port* nối tiếp của 8051 hoạt động như một bộ thu phát không đồng bộ ( universal asynchronous receiver transmitter ) UART 8-bit có tốc độ baud thay đổi. UART là một bộ thu và phát dữ liệu nối tiếp với mỗi một ký tự dữ liệu được đứng trước bởi một bit start ( logic 0 ) và được đứng sau bởi một bit stop ( logic 1 ). Thỉnh thoảng một bit chẵn lẻ được chèn giữa bit dữ liệu sau cùng và bit stop. Hoạt động chủ yếu của một UART là biến đổi dữ liệu phát từ song song thành nối tiếp và biến đổi dữ liệu thu từ nối tiếp thành song song.

Như vậy ở chế độ 1 ta có 10 bit được thu trên chân RxD và 10 bit được phát trên chân TxD cho mỗi một ký tự dữ liệu, chúng bao gồm 1 bit start ( luôn luôn là 0 ), 8 bit dữ liệu ( bit LSB trước tiên ) và 1 bit stop ( luôn luôn là 1 ). Khi hoạt động thu, bit stop đưa đến bit RB8 của SCON. Với 8051, tốc độ baud được thiết lập bởi tốc độ tràn ( overflow rate ) của bộ định thời 1 còn ở 8052, tốc độ baud được thiết lập bởi tốc

độ tràn của bộ định thời 1 hoặc bộ định thời 2 hoặc tổ hợp của cả hai ( một cho phát và một cho thu ).

Việc cấp xung *clock* dịch bit và đồng bộ các thanh ghi dịch bit của *port* nối tiếp ở các chế độ 1, 2 và 3 được thiết lập bởi một bộ đếm 16, ngõ ra của bộ đếm là xung *clock* tốc độ baud ( xem hình 5.5 ). Ngõ vào của bộ đếm vừa nêu được chọn bằng phần mềm và sẽ được trình bày sau.



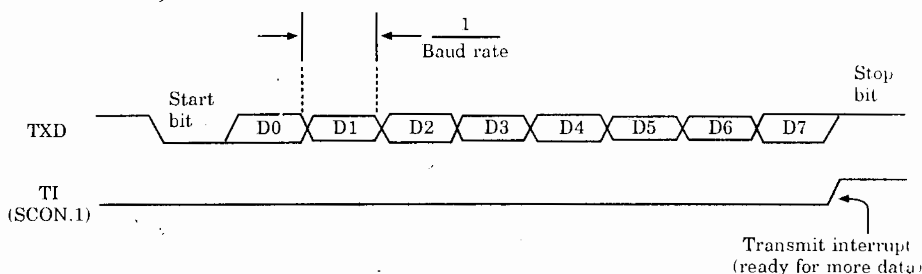

Hình 5.5 : Cấp xung *clock* cho *port* nối tiếp

16 x baud rate : 16 x tốc độ baud

Baud rate clock : xung *clock* tốc độ baud

Serial port shift register : thanh ghi dịch bit của *port* nối tiếp

Việc phát được khởi động bằng cách ghi vào SBUF nhưng việc phát không thực sự bắt đầu cho đến lần tràn kế của bộ đếm 16, bộ đếm cung cấp tốc độ baud cho *port* nối tiếp. Dữ liệu được dịch bit để được xuất ra trên đường TxD sẽ bắt đầu bằng bit start, tiếp theo là 8 bit dữ liệu rồi đến bit stop. Thời gian của mỗi một bit là giá trị nghịch đảo của tốc độ baud, tốc độ baud có được bằng cách lập trình cho bộ định thời. Cờ ngắt phát TI được *set* bằng 1 ngay khi bit stop xuất hiện trên đường TxD ( xem hình 5.6 ).




Hình 5.6 : *Set* cờ TI của *port* nối tiếp

Baud rate : tốc độ baud

Start bit : bit start

Stop bit : bit stop

Transmit interrupt : ngắt phát

Việc nhận được khởi động bởi một chuyển trạng thái từ 1 xuống 0 trên đường RxD ( bắt đầu bit start ). Bộ đếm 16 ngay lập tức được xóa để gán các số đếm cho dòng bit đến chân RxD ( bit kế tiếp đến khi bộ đếm tràn lần nữa và v.v... ). Dòng bit đến được lấy mẫu ở giữa 16 số đếm.

Bộ thu bao gồm việc phát hiện bit start sai bằng cách yêu cầu 8 số đếm ở trạng thái 0 sau khi có sự chuyển trạng thái từ 1 xuống 0 lần đầu tiên. Nếu điều này không xảy ra, bộ thu được giả sử rằng đã được nhận được nhiễu thay vì là nhận một bit hợp lệ. Bộ thu sẽ được thiết lập lại, quay về trạng thái nghỉ và chờ sự chuyển trạng thái từ 1 xuống 0 kế. Giả sử một bit start hợp lệ được phát hiện, việc nhận ký tự sẽ tiếp tục. Bit start được bỏ qua và 8 bit dữ liệu được nhận tuần tự vào thanh ghi dịch bit của *port* nối tiếp. Khi cả 8 bit đã được nhận, các điều sau sẽ xảy ra :

1. Bit thứ 9 ( bit stop ) được đưa đến bit RB8 trong thanh ghi SCON.
2. 8 bit dữ liệu được nạp vào SBUF.
3. Cờ ngắt thu RI được *set*.

Tuy nhiên các điều trên chỉ xảy ra nếu các điều kiện sau tồn tại :

1. RI = 0
2. SM2 = 1 và bit stop nhận được là bit 1, hoặc SM2 = 0.

Yêu cầu RI = 0 đảm bảo rằng phần mềm đã đọc ký tự trước ( và xóa RI ). Điều kiện thứ hai có vẻ phức tạp nhưng chỉ áp dụng trong chế độ truyền thông đa xử lý. Yêu cầu có nghĩa là không *set* RI bằng 1 trong chế độ truyền thông đa xử lý khi bit dữ liệu thứ 9 là 0.

### 5.3.3 UART 9-bit có tốc độ baud cố định ( chế độ 2 )

Khi SM1 = 1 và SM0 = 0, *port* nối tiếp hoạt động ở chế độ 2, chế độ UART 9-bit có tốc độ baud cố định. 11 bit được thu hoặc phát cho việc thu phát một ký tự dữ liệu : bit start, 8 bit dữ liệu, bit dữ liệu thứ 9 lập trình được và bit stop. Khi phát, bit thứ 9 là bit bất kỳ được đặt vào bit TB8 trong thanh ghi SCON ( có thể là bit chẵn lẻ ). Khi thu, bit thứ 9 nhận được sẽ đặt vào bit RB8. Tốc độ baud ở chế độ 2 bằng 1/32 hoặc bằng 1/64 tần số của mạch dao động trên *chip* ( xem mục 5.6 ).

### 5.3.4 UART 9-bit có tốc độ baud thay đổi ( chế độ 3 )

Chế độ 3, UART 9-bit có tốc độ thay đổi, tương tự như chế độ 2 ngoại trừ tốc độ baud được lập trình và được cung cấp bởi bộ định thời. Thật

ra các chế độ 1, 2 và 3 đều tương tự nhau. Chúng khác nhau ở tốc độ baud ( cố định ở chế độ 2 và thay đổi ở các chế độ 1 và 3 ) và ở số bit dữ liệu ( 8 bit ở chế độ 1 và 9 bit ở các chế độ 2 và 3 ).

## 5.4 KHỞI ĐỘNG VÀ TRUY XUẤT CÁC THANH GHI

### 5.4.1 Cho phép thu

Bit cho phép thu REN trong thanh ghi SCON phải được *set* bằng 1 bởi phần mềm để cho phép nhận các ký tự. Điều này thường được thực hiện ở đầu chương trình khi *port* nối tiếp, các bộ định thời v.v... được khởi động và có thể được thực hiện theo 2 cách. Lệnh :

```
SETB  REN
```

*set* bit REN bằng 1 hoặc lệnh :

```
MOV  SCON, #xxx1xxxxB
```

*set* bit REN bằng 1 và xóa hoặc *set* các bit khác trong SCON nếu cần.

### 5.4.2 Bit dữ liệu thứ 9

Bit dữ liệu thứ 9 được phát ở các chế độ 2 và 3 phải được nạp cho bit TB8 bằng phần mềm. Bit dữ liệu thứ 9 thu được phải đặt vào bit RB8 của SCON. Phần mềm có thể yêu cầu hoặc không yêu cầu bit dữ liệu thứ 9 tùy vào các đặc tính của thiết bị nối tiếp mà với thiết bị này việc truyền dữ liệu được thiết lập ( bit thứ 9 còn đóng vai trò quan trọng trong truyền thông đa xử lý ).

### 5.4.3 Thêm vào bit chẵn lẻ

Bit thứ 9 thường được dùng làm bit chẵn lẻ cho một ký tự. Như đã đề cập ở chương 2, bit P trong từ trạng thái chương trình PSW được *set* hoặc xóa ở mỗi một chu kỳ máy để thiết lập việc kiểm tra chẵn cho 8 bit chứa trong thanh chứa A.

Thí dụ nếu việc truyền thông yêu cầu 8 bit dữ liệu cộng với một bit kiểm tra chẵn, các lệnh sau được dùng để phát đi 8 bit trong thanh chứa với bit kiểm tra chẵn được đưa vào bit thứ 9 :

```
MOV  C, P          ; đưa bit kiểm tra chẵn vào TB8
MOV  TB8, C        ; bit này trở thành bit thứ 9
MOV  SBUF, A       ; di chuyển 8 bit dữ liệu từ ACC đến SBUF
```

Nếu kiểm tra lẻ được yêu cầu, các lệnh trên phải được sửa đổi như sau :

```
MOV  C, P          ; đưa bit kiểm tra chẵn vào TB8
CPL  C             ; biến đổi thành kiểm tra lẻ
```

```
MOV  TB8, C         ; bit này trở thành bit thứ 9
MOV  SBUF, A        ; di chuyển 8 bit dữ liệu từ ACC đến SBUF
```

Dĩ nhiên việc sử dụng bit chẵn lẻ không bị giới hạn ở các chế độ 2 và chế độ 3. Trong chế độ 1, 8 bit dữ liệu được phát đi bao gồm 7 bit dữ liệu cộng với bit chẵn lẻ. Để phát đi một mã ASCII 7-bit cùng với bit kiểm tra chẵn ( bit thứ 8 ), ta có thể sử dụng các dòng lệnh sau :

```
CLR  ACC.7          ; đảm bảo bit MSB được xóa
                    ; kiểm tra chẵn
MOV  C, P           ; sao chép bit P vào C
MOV  ACC.7, C       ; đặt bit kiểm tra chẵn vào bit MSB
MOV  SBUF, A        ; phát ký tự
                    ; 7 bit dữ liệu cộng với bit kiểm tra chẵn
```

### 5.4.4 Các cờ ngắt

Các cờ ngắt thu RI và ngắt phát TI trong thanh ghi SCON đóng một vai trò quan trọng trong việc truyền dữ liệu nối tiếp của 8051. Cả 2 bit đều được *set* bằng 1 bằng phần cứng nhưng phải được xóa bằng phần mềm.

Điển hình là RI được *set* bằng 1 khi kết thúc việc nhận một ký tự và chỉ ra rằng bộ đệm thu đầy. Điều kiện này được kiểm tra bằng phần mềm hoặc được lập trình để tạo ra một ngắt ( các ngắt được đề cập trong chương 6 ).

Nếu phần mềm muốn nhập một ký tự từ một thiết bị ghép với *port* nối tiếp, phần mềm phải chờ cho đến khi RI được *set* bằng 1, kế đến phần mềm xóa RI và đọc ký tự từ SBUF. Điều này được thể hiện như sau :

```
WAIT:  JNB   RI, WAIT     ; kiểm tra RI cho đến khi bằng 1
       CLR   RI           ; xóa RI
       MOV   A, SBUF      ; đọc ký tự
```

Cờ TI được *set* bằng 1 khi kết thúc việc phát một ký tự và chỉ ra rằng bộ đệm phát rỗng. Nếu phần mềm muốn phát một ký tự đến một thiết bị ghép với *port* nối tiếp, phần mềm trước tiên phải kiểm tra để biết *port* nối tiếp đã sẵn sàng. Nói cách khác, nếu một ký tự trước đó đã được phát, phần mềm phải chờ việc phát kết thúc trước khi gởi tiếp ký tự kế.

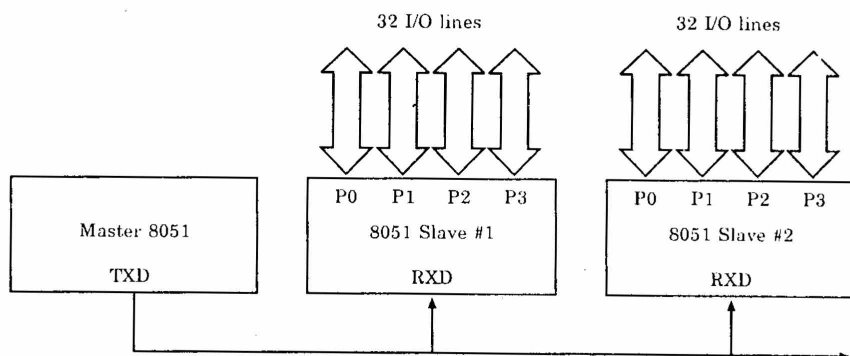
Các lệnh sau đây phát một ký tự chứa trong thanh chứa :

```
WAIT:   JNB   TI, WAIT     ; kiểm tra TI cho đến khi bằng 1
        CLR   TI           ; xóa TI
        MOV   SBUF, A      ; phát ký tự
```

Các chuỗi lệnh thu và phát ở trên là một phần của các chương trình con xuất nhập ký tự. Chúng sẽ được mô tả chi tiết trong thí dụ 5.2 và thí dụ 5.3.

## 5.5 TRUYỀN THÔNG ĐA XỬ LÝ

Các chế độ 2 và 3 là các chế độ dự phòng cho việc truyền thông đa xử lý. Trong các chế độ này, 9 bit dữ liệu được thu và bit thứ 9 được đưa đến RB8. *Port* có thể được lập trình sao cho khi bit stop được nhận, ngắt do *port* nối tiếp được tích cực chỉ nếu RB8 = 1. Đặc trưng này có được bằng cách *set* bit SM2 trong thanh ghi SCON bằng 1. Một ứng dụng cho điều này là một môi trường mạng sử dụng nhiều 8051 được sắp xếp theo mô hình chủ / tớ ( master / slave ) như trình bày trong hình 5.7.




Hình 5.7 : Truyền thông đa xử lý

Master 8051 : 8051 chủ

8051 slave # 1, # 2 : 8051 tớ 1, 2

32 I/O lines : 32 đường xuất/nhập

Khi bộ xử lý chủ muốn truyền một khối dữ liệu đến một trong nhiều bộ xử lý tớ, trước tiên bộ xử lý chủ phát đi một byte địa chỉ nhận dạng bộ xử lý tớ đích. Một byte địa chỉ khác với một byte dữ liệu ở chỗ bit thứ 9 là 1 trong byte địa chỉ và là 0 trong byte dữ liệu. Một byte địa chỉ ngắt tất cả các bộ xử lý tớ để cho mỗi một bộ xử lý tớ có thể khảo sát byte nhận được để kiểm tra xem có phải là bộ xử lý tớ đang được định địa chỉ không. Bộ xử lý tớ được định địa chỉ sẽ xóa bit SM2 của mình và chuẩn

bị nhận các byte dữ liệu theo sau. Các bộ xử lý tớ không được định địa chỉ có các bit SM2 của chúng được *set* bằng 1 và thực thi các công việc của riêng chúng, bỏ qua không nhận các byte dữ liệu. Các bộ xử lý này sẽ được ngắt lần nữa khi bộ xử lý chủ phát tiếp byte địa chỉ kế. Các sơ đồ cụ thể có thể được nêu ra sao cho một khi liên kết chủ tớ đã được thiết lập, bộ xử lý tớ cũng có thể phát đến bộ xử lý chủ. Mưu mẹo ở đây là không sử dụng bit dữ liệu thứ 9 sau khi liên kết vừa được thiết lập ( ngược lại các bộ xử lý tớ khác có thể được chọn một cách không cố ý ).

SM2 không ảnh hưởng đến chế độ 0, và trong chế độ 1 bit này có thể được dùng để kiểm tra sự hợp lệ của bit stop. Ở chế độ 1 thu, nếu SM2 = 1, ngắt thu sẽ không được tích cực trừ phi bit stop thu được là hợp lệ.

## 5.6 TỐC ĐỘ BAUD CỦA *PORT* NỐI TIẾP

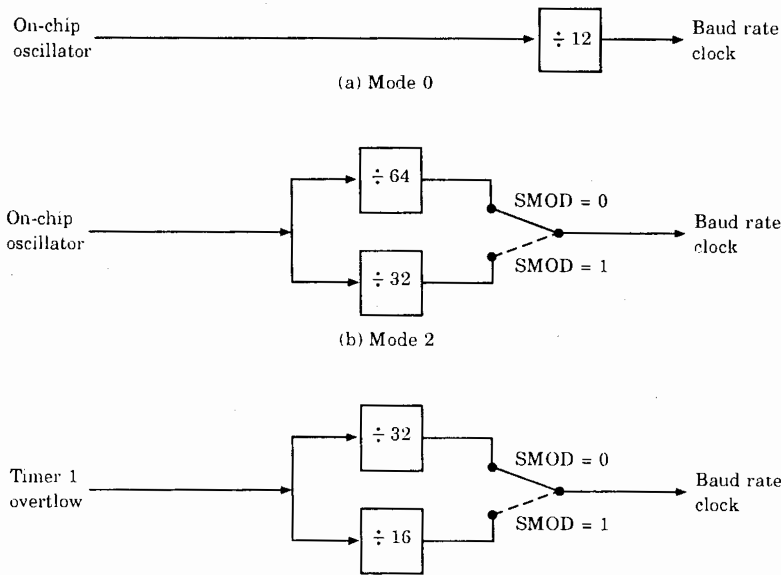
Như đã thấy trong bảng 5.2, tốc độ baud sẽ cố định trong các chế độ 0 và 2. Trong chế độ 0, tốc độ baud luôn luôn bằng tần số của mạch dao động trong *chip* chia cho 12. Thông thường người ta sử dụng một thạch anh bên ngoài *chip* cho mạch dao động này. Giả sử tần số của mạch dao động là 12 MHz, tốc độ baud của chế độ 0 là 1 MHz ( xem hình 5.8a ).

Sau khi hệ thống được *reset*, tốc độ baud của chế độ 2 bằng tần số của mạch dao động chia cho 64. Tốc độ baud cũng bị ảnh hưởng bởi một bit trong thanh ghi điều khiển nguồn PCON. Bit 7 của PCON là bit SMOD và việc *set* bit này bằng 1 sẽ làm tăng tốc độ baud của các chế độ 1, 2 và 3 lên gấp đôi. Ở chế độ 2, tốc độ baud có thể được nhân 2 từ giá trị mặc định là 1/64 tần số của mạch dao động ( SMOD = 0 ) trở thành 1/32 tần số của mạch dao động ( SMOD = 1 ) ( xem hình 5.8b ).

Vì thanh ghi PCON không được định địa chỉ từng bit, việc *set* bit SMOD lên 1 mà không làm thay đổi các bit khác của thanh ghi này được thực hiện bằng những dòng lệnh sau :

```
MOV    A, PCON      ; lấy giá trị hiện hành của PCON
SETB   ACC.7        ; set bit 7 bằng 1 ( SMOD )
MOV    PCON, A      ; ghi giá trị mới vào PCON
```

Các tốc độ baud của 8051 ở chế độ 1 và chế độ 3 được xác định bởi tốc độ tràn của bộ định thời 1. Vì bộ định thời hoạt động ở tần số tương đối cao, ta cần chia tốc độ tràn cho 32 ( hoặc 16 nếu SMOD = 1 ) trước khi trở thành xung *clock* tốc độ baud cung cấp cho *port* nối tiếp. Tốc độ baud của 8052 ở các chế độ 1 và 3 được xác định bởi tốc độ tràn của bộ định thời 1 hoặc bộ định thời 2 hoặc cả hai.




Hình 5.8 : Các nguồn xung *clock* cho *port* nối tiếp (a) chế độ 0 (b) chế độ 2 (c) chế độ 1 và 3

On chip oscillator : bộ dao động trong *chip*

Baud rate clock : xung *clock* tốc độ baud

Timer 1 overflow : tràn bộ định thời 1

### 5.6.1 Sử dụng bộ định thời 1 làm xung *clock* tốc độ baud

Kỹ thuật thường dùng để tạo xung *clock* tốc độ baud là khởi động thanh ghi TMOD ở chế độ tự nạp lại 8-bit ( chế độ định thời 2 ) và đặt giá trị nạp lại thích hợp vào thanh ghi TH1 để có tốc độ tràn đúng, từ đó tạo ra tốc độ baud.

Thanh ghi TMOD được khởi động như sau :

```
MOV     TMOD, #0010xxxxB
```

xxxx dành cho bộ định thời 0.

Đây không phải là khả năng duy nhất. Các tốc độ baud rất chậm có thể nhận được bằng cách sử dụng chế độ 16-bit, chế độ định thời 2 với (TMOD) = 0001xxxxB. Tuy nhiên có một lỗi phần mềm ở đây do các thanh ghi TH1/TL1 phải được khởi động lại sau mỗi lần tràn. Điều này cần được thực hiện trong một trình phục vụ ngắt. Một lựa chọn khác là cung cấp xung *clock* bên ngoài cho bộ định thời 1 bằng cách sử dụng ngõ vào T1 ( P3.5 ). Dù là lựa chọn nào, tốc độ baud cũng bằng tốc độ tràn của bộ định thời 1 chia cho 32 ( hoặc chia cho 16 nếu SMOD = 1 ).

Do vậy, công thức dùng để xác định tốc độ baud ở các chế độ 1 và 3 là :

BAUD RATE = TIMER 1 OVERFLOW RATE ÷ 32

Tốc độ baud = tốc độ tràn bộ định thời 1 chia cho 32

Thí dụ nếu cần tốc độ baud là 1200, tốc độ tràn của bộ định thời 1 phải là 38.4 KHz ( = 1200 x 32 ).

Nếu tần số của mạch dao động bên trong *chip* là 12 MHz, bộ định thời 1 được cấp xung *clock* là 1 MHz hay 1000 KHz. Do bộ định thời phải tràn ở tốc độ là 38.4 KHz, việc tràn cần xảy ra sau mỗi 26.04 xung *clock* ( 1000 ÷ 38.4 ) và được làm tròn là 26. Vì bộ định thời đếm lên và tràn khi có số đếm từ FFH chuyển thành 00H, 26 số đếm nhỏ hơn 0 là giá trị nạp lại cần có để nạp cho thanh ghi TH1. Giá trị này là –26. Cách dễ dàng nhất để đặt giá trị nạp lại vào TH1 là :

```
MOV    TH1, #-26
```

Trình dịch hợp ngữ sẽ thực hiện việc biến đổi cần thiết. Trong trường hợp này –26 được biến đổi thành 0E6H, vậy thì lệnh trên trở thành :

```
MOV    TH1, #0E6H
```

Do ta làm tròn số xung đếm nên sẽ có một sai số nhỏ trong kết quả tính tốc độ baud. Trong trường hợp tổng quát, một sai số 5% là sai số chấp nhận được trong truyền dữ liệu không đồng bộ. Các tốc độ baud chính xác có thể nhận được bằng cách sử dụng một thạch anh 11.059 MHz cho mạch dao động trong *chip*. Bảng 5.3 tóm tắt các giá trị nạp lại cần đưa vào thanh ghi TH1 cho các tốc độ baud thường dùng nhất ( sử dụng thạch anh 12 MHz hoặc 11.059 MHz ).

| **Tốc độ baud** | **Tần số thạch anh** | **SMOD** | **Giá trị nạp cho TH1** | **Tốc độ baud thực tế** | **Sai số** |
|---|---|---|---|---|---|
| 9600 | 12.000 MHz | 1 | –7 (F9H) | 8923 | 7% |
| 2400 | 12.000 MHz | 0 | –13 (F3H) | 2404 | 0.16% |
| 1200 | 12.000 MHz | 0 | –26 (E6H) | 1202 | 0.16% |
| 19200 | 11.059 MHz | 0 | –3 (FDH) | 19200 | 0 |
| 9600 | 11.059 MHz | 0 | –3 (FDH) | 9600 | 0 |
| 2400 | 11.059 MHz | 0 | –12 (F4H) | 2400 | 0 |
| 1200 | 11.059 MHz | 0 | –24 (E8H) | 1200 | 0 |

Bảng 5.3 : Tóm tắt tốc độ baud

**Thí dụ 5.1** : Khởi động *port* nối tiếp

*Viết một chuỗi lệnh để khởi động port nối tiếp sao cho port này hoạt động như một UART 8-bit với tốc độ 2400 baud. Sử dụng bộ định thời 1 để cung cấp xung clock tốc độ baud.*

Với thí dụ này ta phải khởi động 4 thanh ghi : SMOD, TMOD, TCON và TH1. Các giá trị yêu cầu được tóm tắt dưới đây :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | SM0 | SM1 | SM2 | REN | TB8 | RB8 | TI | RI |
| SCON: | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| | GTE | C/T | M1 | M0 | GTE | C/T | M1 | M0 |
| TMOD: | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| | TF1 | TR1 | TF0 | TR0 | IE1 | IT1 | IE0 | IT0 |
| TCON: | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| TH1: | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |

Việc thiết lập SM0/SM1 = 0/1 nhằm đặt *port* nối tiếp ở chế độ UART 8-bit. REN = 1 cho phép *port* nối tiếp thu các ký tự. Việc *set* TI bằng 1 cho phép phát ký tự đầu tiên bằng cách chỉ ra rằng bộ đệm phát rỗng. Với thanh ghi TMOD, việc thiết lập M1/M0 = 1/0 đặt bộ định thời 1 vào chế độ tự nạp lại 8-bit. Việc *set* TR1 trong TCON bằng 1 sẽ khởi động bộ định thời 1 hoạt động.

Các bit khác được cho bằng 0 do chúng điều khiển các đặc trưng hoặc các chế độ không được sử dụng trong thí dụ này.

Giá trị cần nạp cho TH1 là giá trị cung cấp tốc độ tràn : 2400 x 32 = 76.8 KHz. Giả sử tần số của mạch dao động trong *chip* là 12 MHz, bộ định thời 1 được cung cấp xung *clock* có tần số 1 MHz hay 1000 KHz và số xung *clock* cho mỗi lần tràn là 1000 ÷ 76.8 = 13.02 ( làm tròn là 13 ). Giá trị nạp lại là −13 hoặc 0F3H.

Chuỗi lệnh khởi động *port* nối tiếp như sau :

```
        ORG     8100H
INIT:   MOV     SCON, #52H      ; port nối tiếp, chế độ 1
        MOV     TMOD, #20H      ; bộ định thời 1, chế độ 2
        MOV     TH1, #-13       ; giá trị nạp lại để có 2400 baud
        SETB    TR1             ; bộ định thời 1 hoạt động
        END
```

**Thí dụ 5.2** : Chương trình con xuất ký tự

*Viết một chương trình con gọi là OUTCHR để phát mã ASCII 7-bit chứa trong thanh chứa A ra port nối tiếp của 8051 với bit kiểm tra lẻ là bit thứ 8. Việc trở về từ chương trình con không làm thay đổi nội dung thanh chứa ( nghĩa là thanh chứa có nội dung giống như nội dung trước khi chương trình con được gọi ).*

Thí dụ này và thí dụ kế minh họa hai trong nhiều chương trình con thông dụng nhất trên các hệ máy vi tính có thiết bị đầu cuối ghép qua chuẩn RS-232 : xuất ký tự ( OUTCHR ) và nhập ký tự ( INCHAR ).

```
            ORG     8100H
OUTCHR:     MOV     C, P          ; đặt bit chẵn lẻ vào cờ C
            CPL     C             ; đổi thành lẻ
            MOV     ACC.7, C      ; đưa vào bit 7 của ACC
AGAIN:      JNB     TI, AGAIN     ; bộ đệm phát rỗng ?
                                  ; sai, kiểm tra lại
            CLR     TI            ; đúng, xóa cờ TI và
            MOV     SBUF, A       ; phát 1 ký tự
            CLR     ACC.7
            RET
            END
```

Ba lệnh đầu tiên đặt bit kiểm tra lẻ vào bit 7 của thanh chứa. Do bit P trong từ trạng thái chương trình PSW thiết lập kiểm tra chẵn cho thanh chứa, bit này phải được lấy bù trước khi đặt vào ACC.7. Lệnh JNB tạo ra một vòng lặp chờ để kiểm tra cờ ngắt phát TI cho đến khi cờ này được *set* bằng 1. Khi TI =1 ( do việc phát ký tự trước đó vừa kết thúc ), bit này được xóa và sau đó ký tự trong thanh chứa được ghi vào bộ đệm của *port* nối tiếp SBUF và việc phát ký tự bắt đầu ở lần tràn kế của bộ đếm 16 tạo xung *clock* cho port nối tiếp. Sau cùng bit ACC.7 được xóa để giá trị trả về giống như khi mã 7-bit được chuyển đến chương trình con.

Chương trình con OUTCHR thường được gọi bởi một chương trình gọi để phát một ký tự hoặc một chuỗi ký tự. Thí dụ các lệnh sau phát mã ASCII của ký tự Z đến thiết bị nối tiếp ghép với *port* nối tiếp của 8051 :

```
MOV   A, # 'Z'
CALL  OUTCHR
```

Ta cũng có thể sử dụng chương trình con OUTCHR như là một khối được thiết kế trong một chương trình con có tên là OUTSTR chẳng hạn, chương trình con này phát một chuỗi ký tự ASCII kết thúc bởi byte NULL ( 00H ) đến một thiết bị nối tiếp ghép với *port* nối tiếp của 8051.

**Thí dụ 5.3** : Chương trình con thu một ký tự

*Viết một chương trình con có tên là INCHAR để thu một ký tự từ port nối tiếp của 8051 và trả về mã ASCII 7-bit trong thanh chứa. Sử dụng kiểm tra lẻ trong bit thứ 8 thu được và set cờ nhớ bằng 1 nếu có lỗi chẵn lẻ.*

```
         ORG   8100H
INCHAR:  JNB   RI, $      ; chờ ký tự
         CLR   RI         ; xóa cờ
         MOV   A, SBUF    ; đọc ký tự vào thanh chứa
         MOV   C, P       ; với kiểm tra lẻ trong A
                          ; P cần được set bằng 1
         CPL   C          ; việc lấy bù chỉ ra có lỗi hay
                          ; không ?
         CLR   ACC.7      ; xóa
         RET
         END
```

Chương trình con này bắt đầu bằng việc chờ cờ ngắt thu RI được *set* bằng 1 để chỉ ra rằng ký tự đã sẵn sàng trong bộ đệm thu SBUF ( để được đọc ). Khi RI = 1, lệnh JNB chuyển điều khiển đến lệnh tiếp theo lệnh này. Cờ RI được xóa và mã trong SBUF được đọc vào thanh chứa. Bit P trong PSW thiết lập kiểm tra chẵn cho thanh chứa, do vậy bit này cần được *set* bằng 1 nếu bản thân thanh chứa chứa bit kiểm tra lẻ ở bit thứ 7 của thanh ghi này. Việc di chuyển bit P vào cờ nhớ làm cho CY = 0 nếu không có lỗi. Mặt khác, nếu thanh chứa chứa một lỗi chẵn lẻ, cờ CY sẽ bằng 1. Cuối cùng bit ACC.7 được xóa để đảm bảo rằng chỉ có mã 7-bit được trả về cho chương trình gọi.

Các nội dung trong các chương 4 và 5 trình bày các chi tiết chính cần đến khi ta lập trình định thời và thu phát nối tiếp. Các chương trình thí dụ trong 2 chương này hoàn toàn không sử dụng ngắt mà sử dụng các vòng lặp. Việc sử dụng các vòng lặp trong lập trình xuất nhập

gặp phải khuyết điểm là bộ vi xử lý không làm gì cả để chờ các điều kiện xuất nhập sẵn sàng.

Trên thực tế, nhiều ứng dụng liên quan đến các bộ định thời và *port* nối tiếp của 8051 yêu cầu phải được đồng bộ cũng như để bộ vi xử lý không phải chờ, người ta sử dụng ngắt. Đây là chủ đề của chương tiếp theo, hoạt động ngắt.

# 6

# HOẠT ĐỘNG NGẮT

## 6.1 MỞ ĐẦU

Ngắt ( interrupt ) là sự xảy ra của một điều kiện – một sự kiện – làm cho chương trình hiện hành bị tạm ngưng trong khi điều kiện được phục vụ bởi một chương trình khác. Các ngắt đóng vai trò quan trọng trong việc thiết kế và hiện thực các ứng dụng của bộ vi điều khiển. Các ngắt cho phép hệ thống đáp ứng một sự kiện theo cách không đồng bộ và xử lý sự kiện trong khi một chương trình khác đang thực thi. Một hệ thống được điều khiển bởi ngắt cho ta ảo tưởng đang làm nhiều công việc đồng thời.

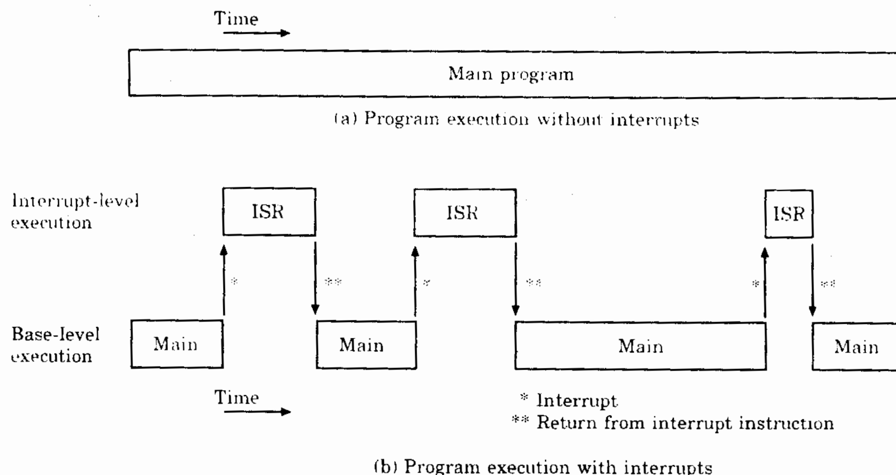
CPU dĩ nhiên không thể thực thi nhiều hơn một lệnh ở một thời điểm nhưng CPU có thể ngưng tạm thời việc thực thi một chương trình để thực thi một chương trình khác rồi sau đó quay trở về thực thi tiếp chương trình đang bị tạm ngưng, điều này giống như CPU rời khỏi chương trình gọi để thực thi chương trình con bị gọi để rồi sau đó quay trở về chương trình gọi. Sự khác nhau của hai vấn đề vừa nêu là trong một hệ thống được điều khiển bởi ngắt, việc ngắt nhằm đáp ứng một sự kiện mà sự kiện này xuất hiện không đồng bộ với chương trình chính đang được thực thi và chương trình chính ( hay nói cách khác là CPU ) không biết trước là sẽ bị ngắt khi nào.

Chương trình xử lý một ngắt được gọi là trình phục vụ ngắt ISR ( interrupt service routine ) hay quản lý ngắt ( interrupt handler ). ISR được thực thi nhằm đáp ứng một ngắt và trong trường hợp tổng quát thực hiện việc xuất nhập đối với một thiết bị. Khi một ngắt xuất hiện, việc thực thi chương trình chính tạm thời bị dừng và CPU thực hiện việc rẽ nhánh đến trình phục vụ ngắt ISR. CPU thực thi ISR để thực hiện một công việc và kết thúc việc thực thi này khi gặp lệnh " quay về từ một trình phục vụ ngắt "; chương trình chính được tiếp tục tại nơi bị tạm dừng. Ta có thể nói chương trình chính được thực thi ở mức nền ( base level ) còn ISR được thực thi ở mức ngắt ( interrupt level ).

Cách nhìn ngắn gọn này được mô tả ở hình 6.1, hình này trình bày :

a- Việc thực thi một chương trình không có ngắt.

b- Việc thực thi ở mức nền có ngắt và các ISR được thực thi ở mức ngắt.




Hình 6.1 : Thực thi chương trình có và không có ngắt (a) không có ngắt (b) có ngắt

Time : thời gian

Main program : chương trình chính

Program execution without interrupts : thực thi chương trình không có ngắt

Program execution with interrupts : thực thi chương trình có ngắt

Interrupt level execution : thực thi ở mức ngắt

Base level execution : thực thi ở mức nền

Interrupt : ngắt

Return from interrupt instruction : trở về từ lệnh ngắt

Một thí dụ về ngắt điển hình là nhập bằng tay sử dụng bàn phím. Ta hãy khảo sát một ứng dụng của lò viba. Chương trình chính có thể điều khiển thành phần công suất của lò để thực hiện việc nấu nướng ; tuy nhiên trong khi đang nấu, hệ thống phải đáp ứng việc nhập bằng tay trên cửa lò, chẳng hạn như một yêu cầu rút ngắn bớt hay kéo dài thêm thời gian nấu. Khi người sử dụng buông phím nhấn, một ngắt được tạo ra ( chẳng hạn một tín hiệu từ mức cao chuyển xuống mức thấp ) và chương trình chính bị ngắt. ISR được thực thi để đọc mã phím và thay đổi các điều kiện nấu tương ứng, sau đó kết thúc bằng cách chuyển điều khiển trở về chương trình chính.

Chương trình chính được thực thi tiếp từ nơi tạm dừng. Điều quan trọng trong thí dụ trên là việc nhập bằng tay xuất hiện không đồng bộ nghĩa là xuất hiện ở các khoảng thời gian không báo trước hoặc được điều khiển bởi phần mềm đang được thực thi trong hệ thống. Đấy là một ngắt.

## 6.2 TỔ CHỨC NGẮT CỦA 8051

Có 5 nguyên nhân tạo ra ngắt ( gọi tắt là nguyên nhân ngắt ) đối với 8051 : hai ngắt do bên ngoài, hai ngắt do bộ định thời và một ngắt do *port* nối tiếp. 8052 có thêm nguyên nhân ngắt thứ 6 : do bộ định thời được thêm vào, bộ định thời thứ ba. Khi ta thiết lập trạng thái ban đầu cho hệ thống ( gọi tắt là *reset* hệ thống ), tất cả các ngắt đều bị vô hiệu hóa ( cấm ) và sau đó chúng được cho phép riêng rẽ bằng phần mềm.

Khi xảy ra hai hay nhiều ngắt đồng thời hoặc xảy ra một ngắt trong khi một ngắt khác đang được phục vụ, ta có 2 sơ đồ xử lý các ngắt : sơ đồ chuỗi vòng và sơ đồ hai mức ưu tiên. Sơ đồ chuỗi vòng là sơ đồ cố định còn sơ đồ ưu tiên ngắt được lập trình bởi người sử dụng.

Ta sẽ bắt đầu khảo sát cách thức cho phép và không cho phép ngắt.

### 6.2.1 Cho phép và không cho phép ngắt

Mỗi một nguyên nhân ngắt được cho phép hoặc không cho phép riêng rẽ thông qua thanh ghi chức năng đặc biệt định địa chỉ bit, thanh ghi cho phép ngắt IE ( interrupt enable ) có địa chỉ byte là 0A8H. Mỗi một bit của thanh ghi này cho phép hoặc không cho phép từng nguyên nhân ngắt riêng rẽ, thanh ghi IE đồng thời còn có một bit toàn cục ( global ) cho phép hoặc không cho phép tất cả các ngắt ( không cho phép khi bị xóa và cho phép khi được *set* bằng 1 ) ( xem bảng 6.1 ).

| ***Bit*** | ***Ký hiệu*** | ***Địa chỉ Bit*** | ***Mô tả*** **( 0 : không cho phép ; 1 : cho phép )** |
|---|---|---|---|
| IE.7 | EA | AFH | Cho phép / không cho phép toàn cục |
| IE.6 | – | AEH | Không sử dụng |
| IE.5 | ET2 | ADH | Cho phép ngắt do bộ định thời 2 |
| IE.4 | ES | ACH | Cho phép ngắt do *port* nối tiếp |
| IE.3 | ET1 | ABH | Cho phép ngắt do bộ định thời 1 |
| IE.2 | EX1 | AAH | Cho phép ngắt từ bên ngoài ( ngắt ngoài 1 ) |
| IE.0 | EX0 | A8H | Cho phép ngắt từ bên ngoài ( ngắt ngoài 0 ) |
| IE.1 | ET0 | A9H | Cho phép ngắt do bộ định thời 0 |

Bảng 6.1 : Thanh ghi cho phép ngắt IE

Hai bit sau đây phải được *set* bằng 1 để cho phép một ngắt nào đó : bit cho phép ngắt riêng rẽ và bit cho phép ngắt toàn cục. Thí dụ ngắt do bộ định thời 1 được cho phép bằng cách dùng 2 lệnh :

```
SETB   ET1    ; Cho phép ngắt do bộ định thời 1
SETB   EA     ; set bit EA bằng 1 để cho phép ngắt toàn cục
```

hoặc bằng cách dùng lệnh sau :

```
MOV    IE, #10001000B
```

Mặc dù cả hai cách nêu trên đều cho ta cùng một kết quả sau khi hệ thống được thiết lập lại trạng thái ban đầu ( *reset* hệ thống ), ảnh hưởng của 2 cách này có khác nhau vì cách thứ hai ghi lên thanh ghi IE trong khi chương trình đang hoạt động.

Cách thứ nhất không gây ảnh hưởng đến 5 bit còn lại của thanh ghi IE trong khi cách thứ hai sẽ xóa các bit khác. Tốt nhất ta nên khởi động thanh ghi IE bằng lệnh di chuyển byte ở đầu chương trình ngay sau khi hệ thống được thiết lập lại. Việc cho phép và không cho phép các ngắt trong chương trình nên sử dụng các lệnh *set* bit và xóa bit để tránh ảnh hưởng đến các bit khác trong thanh ghi IE.

### 6.2.2 Ưu tiên ngắt

Mỗi một nguyên nhân ngắt được lập trình riêng rẽ để có một trong hai mức ưu tiên thông qua thanh ghi chức năng đặc biệt được định địa chỉ bit, thanh ghi ưu tiên ngắt IP ( interrupt priority ), thanh ghi này có địa chỉ byte là 0B8H ( xem bảng 6.2 ).

| ***Bit*** | ***Ký hiệu*** | ***Địa chỉ bit*** | ***Mô tả*** **(1 : mức cao ; 0 : mức thấp)** |
|---|---|---|---|
| IP.7 | . | . | Không sử dụng |
| IP.6 | . | . | Không sử dụng |
| IP.5 | PT2 | 0BDH | Ưu tiên cho ngắt do bộ định thời 2 |
| IP.4 | PS | 0BCH | Ưu tiên cho ngắt do *port* nối tiếp |
| IP.3 | PT1 | 0BBH | Ưu tiên cho ngắt do bộ định thời 1 |
| IP.2 | PX1 | 0BAH | Ưu tiên cho ngắt do bên ngoài (ngắt ngoài 1) |
| IP.1 | PT0 | 0B9H | Ưu tiên cho ngắt do bộ định thời 0 |
| IP.0 | PX0 | 0B8H | Ưu tiên cho ngắt do bên ngoài (ngắt ngoài 0) |

Bảng 6.2 : Thanh ghi ưu tiên ngắt IP

Khi hệ thống được thiết lập lại trạng thái ban đầu, thanh ghi IP sẽ mặc định đặt tất cả các ngắt ở mức ưu tiên thấp. Ý tưởng " các mức ưu tiên " cho phép một trình phục vụ ngắt được tạm dừng bởi một ngắt khác nếu ngắt mới này có mức ưu tiên cao hơn mức ưu tiên của ngắt hiện đang được phục vụ. Điều này hoàn toàn hợp lý đối với 8051 vì ta chỉ có 2 mức ưu tiên. Nếu có ngắt với ưu tiên cao xuất hiện, trình phục vụ ngắt cho ngắt có mức ưu tiên thấp phải tạm dừng ( nghĩa là bị ngắt ). Ta không thể tạm dừng một chương trình phục vụ ngắt có mức ưu tiên cao.

Chương trình chính do được thực thi ở mức nền và không được kết hợp với một ngắt nào nên luôn luôn bị ngắt bởi các ngắt cho dù các ngắt này có mức ưu tiên thấp hay cao. Nếu có 2 ngắt với mức ưu tiên ngắt khác nhau xuất hiện đồng thời, ngắt có mức ưu tiên cao sẽ được phục vụ trước.

### 6.2.3 Chuỗi vòng

Nếu có 2 ngắt có cùng mức ưu tiên xuất hiện đồng thời, chuỗi vòng cố định sẽ xác định ngắt nào được phục vụ trước. Chuỗi vòng này sẽ là : ngắt ngoài 0, ngắt do bộ định thời 0, ngắt ngoài 1, ngắt do bộ định thời 1, ngắt do *port* nối tiếp, ngắt do bộ định thời 2.

Hình 6.2 minh họa 5 nguyên nhân ngắt, cơ chế cho phép riêng rẽ và toàn cục, chuỗi vòng và các mức ưu tiên. Trạng thái của tất cả các nguyên nhân ngắt được thể hiện thông qua các bit cờ tương ứng trong các thanh ghi chức năng đặc biệt có liên quan. Dĩ nhiên nếu một ngắt nào đó không được phép, nguyên nhân ngắt tương ứng không thể tạo ra một ngắt nhưng phần mềm vẫn có thể kiểm tra cờ ngắt. Lấy thí dụ bộ định thời và *port* nối tiếp trong 2 chương trước sử dụng các cờ ngắt một cách rộng rãi dù không có ngắt tương ứng xảy ra, nghĩa là không sử dụng các ngắt.

Ngắt do *port* nối tiếp là kết quả OR của cờ ngắt khi thu RI ( cờ ngắt thu ) với cờ ngắt khi phát TI ( cờ ngắt phát ). Ngắt do bộ định thời 2 được tạo ra do cờ tràn bộ định thời TF2 hoặc do cờ từ bên ngoài EXF2. Các bit cờ tạo ra các ngắt được tóm tắt ở bảng 6.3.

## 6.3 XỬ LÝ NGẮT

Khi có một ngắt xuất hiện và được CPU chấp nhận, chương trình chính bị ngắt. Các thao tác sau đây xảy ra :

- Hoàn tất việc thực thi lệnh hiện hành.
- Bộ đếm chương trình PC được cất vào *stack*.
- Trạng thái của ngắt hiện hành được lưu giữ lại.

– Các ngắt được chặn lại ở mức ngắt.

– Bộ đếm chương trình PC được nạp địa chỉ vector của trình phục vụ ngắt ISR.

– ISR được thực thi.

ISR được thực thi để đáp ứng công việc của ngắt. Việc thực thi ISR kết thúc khi gặp lệnh RETI ( trở về từ một trình phục vụ ngắt ). Lệnh này lấy lại giá trị cũ của bộ đếm chương trình PC từ *stack* và phục hồi trạng thái của ngắt cũ. Việc thực thi chương trình chính được tiếp tục ở nơi bị tạm ngưng.

| ***Ngắt*** | ***Cờ*** | ***Thanh ghi SFR và vị trí bit*** |
|---|---|---|
| Do bên ngoài ( ngắt ngoài 0 ) | IE0 | TCON.1 |
| Do bên ngoài ( ngắt ngoài 1 ) | IE1 | TCON.3 |
| Do bộ định thời 1 | TF1 | TCON.7 |
| Do bộ định thời 0 | TF0 | TCON.5 |
| Do *port* nối tiếp | TI | SCON.1 |
| Do *port* nối tiếp | RI | SCON.0 |
| Do bộ định thời 2 | TF2 | T2CON.7 ( 8052 ) |
| Do bộ định thời 2 | EXF2 | T2CON.6 ( 8052 ) |

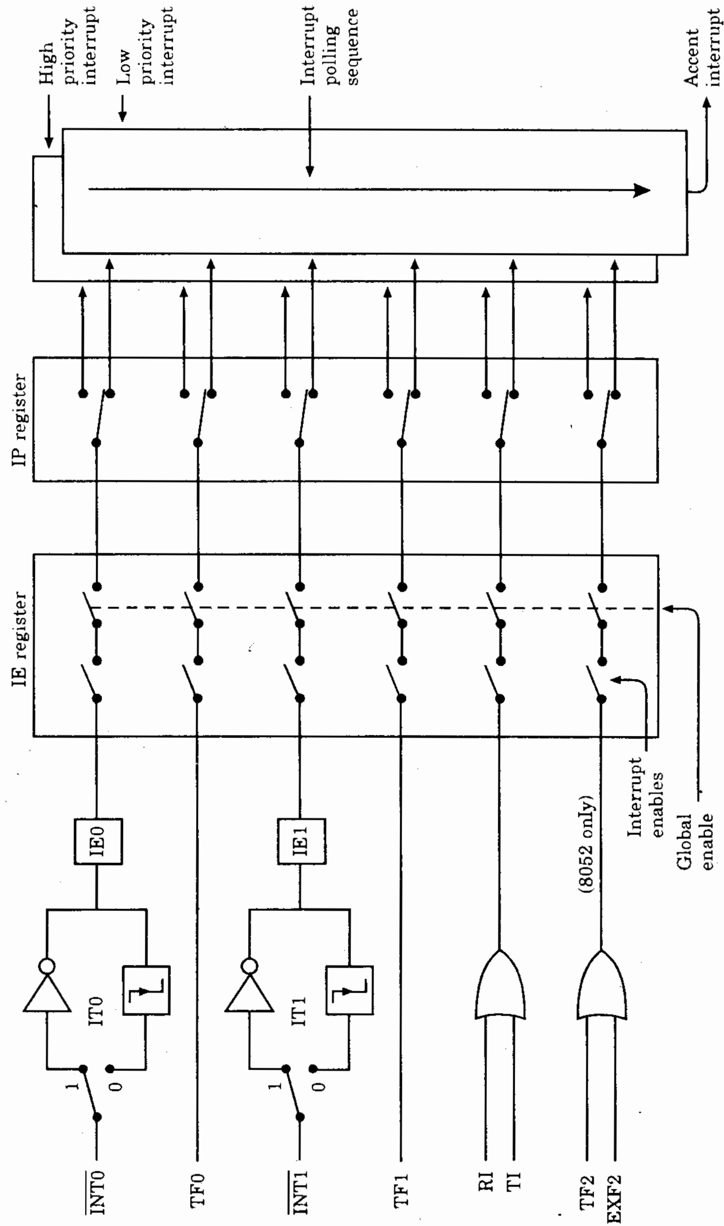
Bảng 6.3 : Các cờ ngắt

### 6.3.1 Các vector ngắt

Khi một ngắt được chấp nhận, giá trị được nạp cho bộ đếm chương trình PC được gọi là vector ngắt. Vector ngắt là địa chỉ bắt đầu của trình phục vụ ngắt của nguyên nhân ngắt tương ứng. Các vector ngắt được cho ở bảng 6.4.

Vector *reset* hệ thống ( RST ở địa chỉ 0000H ) được chứa trong bảng này vì vậy cũng được xem như là 1 ngắt : chương trình chính bị ngắt và bộ đếm chương trình PC được nạp giá trị mới.

Khi một trình phục vụ ngắt được trỏ tới, cờ gây ra ngắt sẽ tự động bị xóa về 0 bởi phần cứng. Các ngoại lệ bao gồm các cờ RI và TI đối với các ngắt do *port* nối tiếp ; TF2 và EXF2 đối với các ngắt do bộ định thời 2. Các nguyên nhân ngắt thuộc 2 ngoại lệ vừa nêu trên do có 2 khả năng tạo ra ngắt nên trong thực tế CPU không xóa cờ ngắt.




Hình 6.2 : Cấu trúc ngắt của 8051

IE register : thanh ghi IE

IP register : thanh ghi IP

High priority interrupt : ngắt ưu tiên cao

Low priority interrupt : ngắt ưu tiên thấp

Interrupt polling sequence : chuỗi vòng ngắt

Interrupt enable : cho phép ngắt

Global enable : cho phép toàn cục

| *Ngắt do* | *Cờ* | *Địa chỉ vector* |
|---|---|---|
| *Reset* hệ thống | RST | 0000H |
| Ngắt ngoài 0 | IE0 | 0003H |
| Bộ định thời 0 | TF0 | 000BH |
| Ngắt ngoài 1 | IE1 | 0013H |
| Bộ định thời 1 | TF1 | 001BH |
| *Port* nối tiếp | RI hoặc TI | 0023H |
| Bộ định thời 2 | TF2 hoặc EXF2 | 002BH |

Bảng 6.4 : Các vector ngắt

Các bit cờ này phải được kiểm tra trong ISR để xác định nguyên nhân ngắt và sau đó cờ gây ra ngắt được xóa bởi phần mềm. Thông thường sẽ có một rẽ nhánh chương trình đến công việc tương ứng tùy thuộc vào nguyên nhân ngắt. Vì các vector ngắt đặt ở đáy của bộ nhớ chương trình, lệnh đầu tiên của chương trình chính thường là một lệnh nhảy qua khỏi vùng nhớ chứa các vector ngắt chẳng hạn như lệnh LJMP 0030H.

## 6.4 THIẾT KẾ CHƯƠNG TRÌNH SỬ DỤNG CÁC NGẮT

Các thí dụ trong chương 4 và chương 5 không sử dụng các ngắt mà sử dụng rộng rãi các vòng lặp chờ để kiểm tra các cờ tràn của bộ định thời ( TF0, TF1 và TF2 ) hoặc các cờ phát, cờ thu của *port* nối tiếp ( TI hoặc RI ). Cách này đưa đến vấn đề là thời gian thực thi chương trình của CPU hoàn toàn tiêu phí vào việc chờ các cờ vừa nêu trên được *set* bằng 1. Điều này không thích hợp với các ứng dụng hướng điều khiển trong đó bộ vi điều khiển phải tác động qua lại với nhiều thiết bị xuất nhập đồng thời.

Trong mục này, các thí dụ được phát triển để minh họa các phương pháp thực tế được dùng nhằm hiện thực phần mềm cho các ứng dụng

hướng điều khiển. Thành phần chủ yếu là ngắt. Mặc dù các thí dụ này không nhất thiết phải lớn hơn nhưng chúng sẽ phức tạp hơn và để nhận ra điều này, chúng ta sẽ tiến hành từng bước ở từng thời điểm một. Phương pháp tốt nhất được khuyến cáo là nên theo dõi từ từ các thí dụ này và nên khảo sát phần mềm một cách tỉ mỉ.

Các lỗi thường xảy ra trong các thiết kế hệ thống thường liên quan đến các ngắt. Vì chúng ta đang sử dụng các ngắt, các thí dụ sẽ hoàn chỉnh và được thực thi độc lập. Mỗi một chương trình bắt đầu ở địa chỉ 0000H với giả thiết là chương trình bắt đầu được thực thi sau khi hệ thống được *reset*. Ý tưởng cuối cùng là các chương trình này phát triển cho các ứng dụng chính thức, chúng được thường trú trong ROM hoặc EPROM.

Khuôn mẫu đề nghị cho một chương trình được thực thi độc lập có sử dụng ngắt như sau :

```
        ORG    0000H      ; điểm nhập sau khi reset
        LJMP   MAIN
               .          ; các điểm nhập của ISR
               .
               .
        ORG    0030H      ; điểm nhập của chương trình chính
MAIN :         .          ; chương trình chính bắt đầu
               .
               .
```

Lệnh đầu tiên nhảy đến địa chỉ 0030H ngay trên các vector ngắt nơi các ISR bắt đầu, như được cho ở bảng 6.4.

Hình 6.3 cho ta thấy chương trình chính bắt đầu ở địa chỉ 0030H.

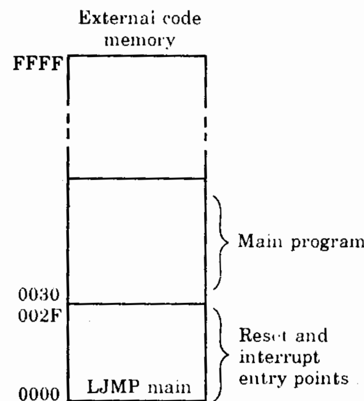
### 6.4.1 Các trình phục vụ ngắt kích thước nhỏ

Các trình phục vụ ngắt phải được bắt đầu ở gần đáy của bộ nhớ chương trình tại các địa chỉ cho ở bảng 6.4. Mặc dù chỉ có 8 byte giữa các điểm nhập của các trình phục vụ ngắt, dung lượng này thường đủ để thực hiện các công việc được yêu cầu và quay trở về chương trình chính từ một trình phục vụ ngắt. Điều này có nghĩa là trình phục vụ ngắt cho các ngắt tương ứng thường không dài quá 8 byte.

Nếu chỉ có một nguyên nhân ngắt được dùng, thí dụ ngắt do bộ định thời 0, thì khuôn mẫu trình bày dưới đây có thể được sử dụng :

```
        ORG   0000H       ; reset
        LJMP  MAIN
        ORG   000BH       ; Điểm nhập của ngắt do bộ định thời 0
T0ISR :       .           ; Bắt đầu ISR cho bộ định thời 0
              .
        RETI              ; trở về chương trình chính
MAIN :        .           ; chương trình chính
              .
```

Nếu có nhiều ngắt được sử dụng, ta phải cẩn thận để đảm bảo các ISR được bắt đầu đúng vị trí và không tràn sang ISR kế. Vì chỉ có một ngắt được sử dụng trong thí dụ trên, chương trình chính có thể bắt đầu ngay sau lệnh RETI.




Hình 6.3 : Tổ chức bộ nhớ khi sử dụng ngắt

External code memory : bộ nhớ chương trình ngoài

Main program: chương trình chính

Reset and interrupt entry points : các điểm nhập của *reset* hệ thống và các ngắt

### 6.4.2. Các trình phục vụ ngắt kích thước lớn

Nếu một trình phục vụ ngắt dài hơn 8 byte được cần đến, ta phải di chuyển chương trình này đến một nơi khác trong bộ nhớ chương trình

hoặc ta có thể cho lấn qua điểm nhập của ISR kế. Điển hình là ISR bắt đầu với một lệnh nhảy đến một vùng khác của bộ nhớ chương trình, ở đó ISR được trải rộng nếu cần. Nếu chỉ khảo sát bộ định thời 0, khuôn mẫu sau đây có thể được sử dụng :

```
        ORG   0000H      ; điểm nhập reset
        LJMP  MAIN
        ORG   000BH      ; điểm nhập của bộ định thời 0
        LJMP  T0ISR
        ORG   0030H      ; phía trên các vector ngắt
MAIN :        .
              .
T0ISR :       .          ; ISR của bộ định thời 0
              .
        RETI             ; quay về chương trình chính
```

Để đơn giản, các chương trình của chúng ta sẽ chỉ làm một việc ở thời điểm bắt đầu. Chương trình chính khởi động bộ định thời, *port* nối tiếp và các thanh ghi ngắt sao cho thích hợp và rồi không làm gì cả. Công việc hoàn toàn được thực hiện bên trong ISR. Sau các lệnh khởi động, chương trình chính chứa lệnh sau :

```
HERE : SJMP  HERE
```

Khi có 1 ngắt xuất hiện, chương trình chính tạm thời bị ngắt trong khi ISR được thực thi. Lệnh RETI ở cuối ISR trả điều khiển về chương trình chính và chương trình này tiếp tục không làm gì cả. Điều này không có gì là không tự nhiên đối với chúng ta. Trong nhiều ứng dụng hướng điều khiển, phần lớn công việc được thực hiện trong trình phục vụ ngắt.

**Thí dụ 6.1** : Tạo sóng vuông sử dụng các ngắt do bộ định thời.

*Viết một chương trình sử dụng bộ định thời 0 và các ngắt để tạo ra một sóng vuông tần số 10KHz trên chân P1.0.*

Các ngắt do bộ định thời xuất hiện khi các thanh ghi định thời ( TLx / THx ) tràn và *set* cờ tràn TFx bằng 1. Thí dụ này đã có ở chương 4 nhưng không sử dụng các ngắt. Phần lớn chương trình sẽ giống như chương trình ở chương 4 ngoại trừ bây giờ chương trình được tổ chức

theo khuôn mẫu của các chương trình có sử dụng ngắt. Dưới đây là chương trình cho thí dụ 6.1

```
        ORG   0              ; điểm nhập reset
        LJMP  MAIN           ; nhảy qua khỏi các vector ngắt
        ORG   000BH          ; vector ngắt của bộ định thời 0
T0ISR:  CPL   P1.0           ; lấy bù
        RETI
        ORG   0030H          ; điểm nhập của chương trình chính
MAIN:   MOV   TMOD, #02H     ; chế độ 2 của bộ định thời 0
        MOV   TH0, #-50      ; trì hoãn 50 µs
        SETB  TR0            ; bộ định thời hoạt động
        MOV   IE, #82H       ; cho phép ngắt do bộ định thời 0
        SJMP  $              ; không làm gì
        END
```

Đây là một chương trình hoàn chỉnh và ta có thể nạp cho EPROM sau khi dịch sang mã máy. Ngay sau khi *reset* hệ thống, bộ đếm chương trình PC được nạp 0000H. Lệnh đầu tiên được thực thi là LJMP MAIN, lệnh này rẽ nhánh đến chương trình chính ở địa chỉ 0030H trong bộ nhớ chương trình. Ba lệnh đầu tiên của chương trình chính khởi động bộ định thời 0 ở chế độ tự nạp lại 8-bit sao cho sẽ tràn sau mỗi 50 µs. Lệnh MOV IE, #82H cho phép các ngắt do bộ định thời 0 tạo ra. Mỗi một lần tràn, bộ định thời sẽ tạo ra một ngắt. Dĩ nhiên lần tràn đầu tiên sẽ không xuất hiện sau 50 µs do chương trình chính đang ở trong vòng lặp " không làm gì ". Khi ngắt xuất hiện sau mỗi 50 µs, chương trình chính bị ngắt và trình phục vụ ngắt cho bộ định thời 0 được thực thi. Ở thí dụ trên trình này chỉ đơn giản lấy bù bit của *port* và quay trở về chương trình chính nơi vòng lặp " không làm gì " được thực thi để chờ một ngắt mới sau 50 µs.

Lưu ý là cờ tràn của bộ định thời TF0 không cần được xóa bởi phần mềm do khi các ngắt được cho phép, cờ này tự động được xóa bởi phần cứng khi CPU trỏ tới trình phục vụ ngắt.

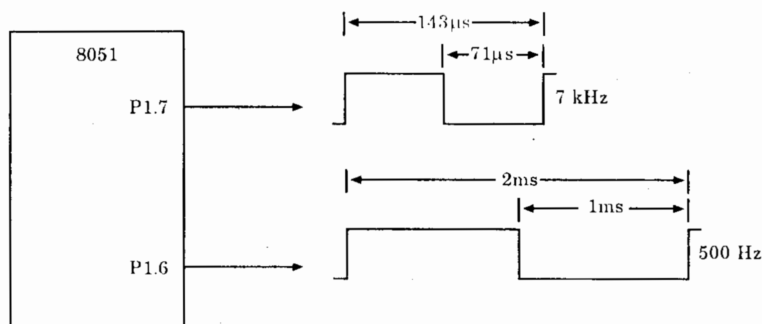
Hiển nhiên địa chỉ quay về trong chương trình là địa chỉ của lệnh SJMP. Địa chỉ này được cất vào vùng *stack* nội của 8051 trước khi CPU trỏ tới trình phục vụ ngắt và được lấy lại từ *stack* khi lệnh RETI ở cuối

trình phục vụ ngắt được thực thi. Vì con trỏ *stack* SP không được khởi động, vùng *stack* được mặc định ở địa chỉ 07H trong RAM nội. Việc cất vào *stack* sẽ cất địa chỉ quay về ở các địa chỉ 08H và 09H trong RAM nội.

**Thí dụ 6.2** : Tạo 2 dạng sóng vuông sử dụng các ngắt.

*Viết một chương trình sử dụng các ngắt để tạo đồng thời các dạng sóng vuông có tần số là 7 KHz và 500 Hz trên các chân P1.7 và P1.6.*

Cấu hình phần cứng cùng các giản đồ thời gian cho các dạng sóng yêu cầu được trình bày ở hình 6.4.




Hình 6.4 : Các dạng sóng của thí dụ 6.2

Tổ hợp các ngõ ra này rất khó tạo ra được trên một hệ thống không được điều khiển ngắt. Bộ định thời 0 hoạt động ở chế độ 2 được sử dụng để tạo ra dạng sóng 7 KHz trên chân P1.7, còn bộ định thời 1 hoạt động ở chế độ 1, chế độ định thời 16-bit, tạo ra dạng sóng 500 Hz trên chân P1.6. Vì dạng sóng 500Hz yêu cầu thời gian mức cao là 1 ms và thời gian mức thấp là 1 ms, chế độ 2 không sử dụng được ( nhắc lại là 256 µs là khoảng thời gian lớn nhất định thời được ở chế độ 2 khi 8051 hoạt động ở tần số 12 MHz ). Chương trình của thí dụ 6.2 như sau :

```
        ORG   0
        LJMP  MAIN
        ORG   000BH              ; địa chỉ vector của bộ định thời 0
        LJMP  T0ISR
        ORG   001BH              ; địa chỉ vector của bộ định thời 1
        LJMP  T1ISR
        ORG   0030H
MAIN :  MOV   TMOD, #12H         ; bộ định thời 1 : chế độ 1
```

```
                                        ; bộ định thời 0 : chế độ 2
         MOV    TH0, #-71               ; 7 KHz sử dụng bộ định thời 0
         SETB   TR0
         SETB   TF1                     ; buộc ngắt do bộ định thời 1
         MOV    IE, #8AH                ; cho phép ngắt do các bộ định thời
         SJMP   $
T0ISR :  CPL    P1.7
         RETI
T1ISR :  CLR    TR1
         MOV    TH1, #HIGH(-1000)       ; thời gian mức cao 1 ms
         MOV    TL1, #LOW(-1000)        ; thời gian mức thấp 1 ms
         SETB   TR1
         CPL    P1.6
         RETI
         END
```

Chương trình cho thí dụ 6.2 cũng là một chương trình hoàn chỉnh có thể được cài đặt trong ROM hoặc EPROM trên sản phẩm sử dụng 8051. Chương trình chính và các trình phục vụ ngắt được đặt bên trên các vector ngắt của *reset* hệ thống và của các ngắt khác. Cả hai dạng sóng được tạo ra bởi các lệnh " CPL bit ", tuy nhiên các khoảng thời gian được định thời cần phải có các phương pháp giải quyết hơi khác nhau một chút.

Do bởi các thanh ghi TL1/TH1 phải được nạp lại sau mỗi lần tràn ( nghĩa là sau mỗi một ngắt ), trình phục vụ ngắt cho bộ định thời 1 :

a) dừng bộ định thời.

b) nạp lại cho TL1/TH1.

c) bắt đầu bộ định thời.

d) lấy bù bit của *port*.

Cũng cần chú ý là các thanh ghi TL1/TH1 không được khởi động ở đầu chương trình chính, khác với TH0. Do TL1/TH1 phải được nạp lại sau mỗi lần tràn bộ định thời, TF1 được *set* bằng 1 trong chương trình chính bởi phần mềm để buộc phải có một ngắt ban đầu ngay trước khi

các ngắt được cho phép. Điều này có hiệu quả cho việc bắt đầu dạng sóng 500Hz.

Trình phục vụ ngắt cho bộ định thời 0 cũng như trong thí dụ trước, chỉ đơn giản lấy bù bit của *port* và quay trở về chương trình chính. SJMP $ được sử dụng trong chương trình chính là dạng viết tắt của HERE : SJMP HERE; hai dạng này có cùng chức năng ( xem mục các ký hiệu của hợp ngữ trong chương 7 ).

## 6.5 CÁC NGẮT DO *PORT* NỐI TIẾP

Các ngắt do *port* nối tiếp xuất hiện khi cờ ngắt phát TI hoặc cờ ngắt thu RI được *set* bằng 1. Một ngắt phát xuất hiện khi việc phát một ký tự đã ghi vào SBUF hoàn tất. Một ngắt thu xuất hiện khi một ký tự được thu nhận đầy đủ và đang ở trong SBUF để chờ được đọc. Như vậy ngắt phát xảy ra khi bộ đệm phát SBUF rỗng còn ngắt thu xảy ra khi bộ đệm thu SBUF đầy.

Các ngắt do *port* nối tiếp có khác với các ngắt do bộ định thời. Cờ gây ra ngắt ở *port* nối tiếp không được xóa bởi phần cứng khi CPU trỏ tới trình phục vụ ngắt. Lý do là vì ở đây ta có 2 nguyên nhân tạo ra ngắt ở *port* nối tiếp, cụ thể là 2 ngắt tạo ra bởi 2 cờ TI và RI ( cờ ngắt phát và cờ ngắt thu ). Nguyên nhân ngắt phải được xác định trong trình phục vụ ngắt và cờ tạo ra ngắt được xóa bởi phần mềm. Cần nhắc lại với các ngắt do bộ định thời, cờ tạo ra ngắt được xóa bởi phần cứng khi CPU trỏ tới trình phục vụ ngắt.

**Thí dụ 6.3** : Xuất ký tự sử dụng ngắt.

*Viết 1 chương trình sử dụng các ngắt để liên tục phát đi tập mã ASCII ( bao gồm cả các mã điều khiển ) đến 1 thiết bị đầu cuối nối với 8051 qua port nối tiếp.*

Có 128 mã ASCII 7-bit trong bảng mã ASCII. Các mã này bao gồm 95 mã đồ họa ( từ 20H đến 7EH ) và 33 mã điều khiển ( từ 00H đến 1FH và 7FH ). Chương trình dưới đây là 1 chương trình hoàn chỉnh và được thực thi từ ROM hoặc EPROM ngay sau khi *reset* hệ thống.

Sau khi nhảy đến nhãn MAIN ở địa chỉ 0030H, ba lệnh đầu tiên khởi động bộ định thời 1 để cung cấp xung *clock* 1200 baud cho *port* nối tiếp, lệnh MOV SCON, #42H khởi động *port* nối tiếp ở chế độ 1 ( UART 8-bit ) và *set* cờ TI bằng 1 để buộc tạo ra một ngắt trước khi các ngắt được cho phép. Sau đó mã đồ họa ASCII đầu tiên ( 20H ) được nạp cho thanh ghi A và các ngắt do *port* nối tiếp được cho phép. Cuối cùng phần chính của chương trình đi vào vòng lặp " không làm gì " (lệnh SJMP $).

```
         ORG    0
         LJMP   MAIN
         ORG    0023H          ; điểm nhập của ngắt do port nối tiếp
         LJMP   SPISR
         ORG    0030H
MAIN :   MOV    TMOD, #20H     ; bộ định thời 1 : chế độ 2
         MOV    TH1, #-26      ; giá trị nạp lại : 1200 baud
         SETB   TR1            ; bộ định thời hoạt động
         MOV    SCON, #42      ; chế độ 1, set TI bằng 1 để buộc
                               ; có ngắt đầu tiên ; gởi ký tự thứ nhất.
         MOV    A, #20H        ; gởi ký tự trắng đầu tiên.
         MOV    IE, #90H       ; cho phép ngắt do port nối tiếp
         SJMP   $              ; không làm gì.
SPISR:   CJNE   A, #7FH, SKIP  ; nếu kết thúc tập mã ASCII
         MOV    A, #20H        ; reset đến SPACE
SKIP:    MOV    SBUF, A        ; gởi ký tự đến port nối tiếp
         INC    A              ; ký tự kế
         CLR    TI             ; xóa cờ ngắt phát
         RETI
         END
```

Trình phục vụ ngắt của *port* nối tiếp làm tất cả công việc một khi chương trình chính đã thiết lập các điều kiện ban đầu. Hai lệnh đầu tiên kiểm tra thanh chứa và nếu mã ASCII đạt đến 7FH ( nghĩa là mã vừa mới được phát đi là 7EH ), thanh chứa được thiết lập lại với nội dung là 20H. Sau đó mã ASCII được gởi đến bộ đệm của *port* nối tiếp ( MOV SBUF, A ), tăng thanh chứa A để có mã kế, cờ ngắt phát được xóa ( CLR TI ) và trình phục vụ ngắt kết thúc ( RETI ). Điều khiển trả về chương trình chính và lệnh SJMP $ được thực thi cho đến khi TI lại được *set* bằng 1.

Nếu ta so sánh tốc độ của CPU với tốc độ truyền ký tự, ta thấy rằng lệnh SJMP $ được thực thi với phần trăm tỉ lệ thời gian rất lớn trong chương trình này. Phần trăm tỉ lệ này bằng bao nhiêu ? Ở tốc độ 1200 baud, mỗi một bit được truyền trong một khoảng thời gian là 1/1200 = 0,833 ms. Như vậy 8 bit dữ liệu cộng với 1 bit start và 1 bit stop chiếm

8,33 ms hay 8333 μs. Thời gian thực thi tệ nhất của trình phục vụ ngắt SPISR là tổng của số chu kỳ cho mỗi một lệnh nhân với 1 μs. Số tính được là 8 μs. Do vậy, với 8333 μs dùng để truyền 1 ký tự chỉ có 8 μs dành cho trình phục vụ ngắt. Lệnh SJMP $ thực thi trong khoảng ( 8325/8333 ) x 100 = 99,9% thời gian. Do ngắt được sử dụng, lệnh SJMP $ có thể được thay bởi các lệnh khác để thực hiện những công việc khác theo yêu cầu của ứng dụng. Các ngắt vẫn xuất hiện và các ký tự vẫn được phát từ *port* nối tiếp sau mỗi một 8,33 ms.

## 6.6 CÁC NGẮT NGOÀI

Ngắt ngoài xảy ra khi có mức thấp hoặc có cạnh âm trên chân $\overline{INT0}$ hoặc $\overline{INT1}$ của 8051. Đây là các chân đa hợp với 2 chân P3.2 và P3.3 của *port* 3.

Thực tế các cờ tạo ra các ngắt này là các bit IE0 và IE1 của thanh ghi TCON. Khi một ngắt ngoài được tạo ra, cờ tạo ra ngắt được xóa bởi phần cứng khi CPU trỏ đến trình phục vụ ngắt chỉ nếu ngắt thuộc loại tác động cạnh. Nếu ngắt thuộc loại tác động mức, nguyên nhân ngắt ngoài sẽ điều khiển mức của cờ thay vì là phần cứng trên chip.

Việc chọn các ngắt loại tác động cạnh hay các ngắt loại tác động mức được lập trình thông qua các bit IT0 và IT1 của thanh ghi TCON. Lấy thí dụ nếu IT1 = 0, ngắt ngoài 1 được kích khởi bởi việc phát hiện mức thấp ở chân $\overline{INT1}$. Nếu IT1 = 1, ngắt ngoài 1 được kích khởi cạnh. Ở chế độ này, nếu các mẫu liên tiếp ở chân $\overline{INT1}$ cho thấy chân này ở mức cao trong một chu kỳ và ở mức thấp trong chu kỳ kế, cờ ngắt IE1 trong thanh ghi TCON được *set* bằng 1 ; kế đến IE1 yêu cầu một ngắt.

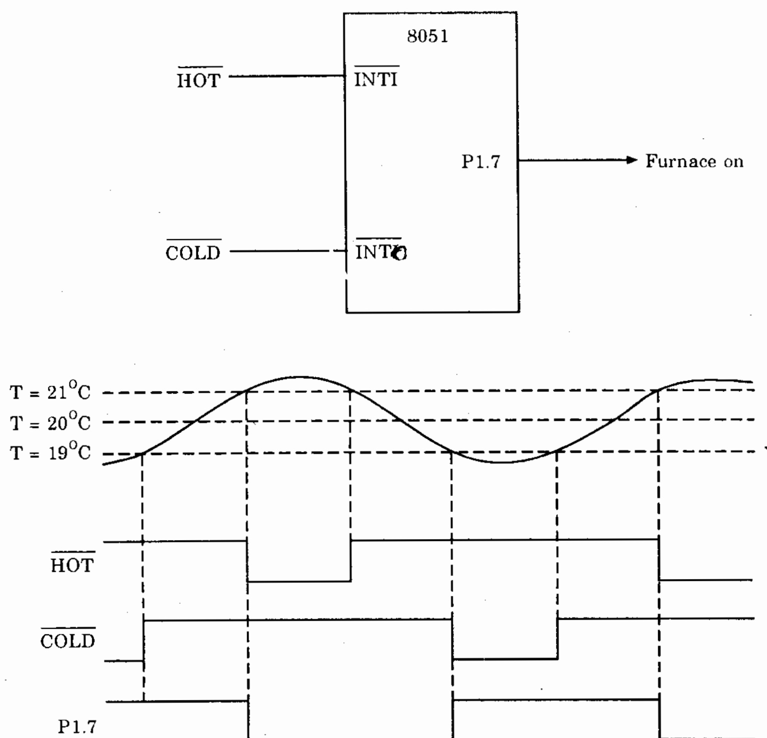
Vì các chân ngắt ngoài được lấy mẫu một lần ở mỗi một chu kỳ máy, các ngõ vào này phải được duy trì tối thiểu 12 chu kỳ dao động để đảm bảo rằng việc lấy mẫu là đúng. Nếu ngắt ngoài thuộc loại tác động cạnh, nguyên nhân ngắt ngoài phải được duy trì tại chân yêu cầu ở mức cao tối thiểu một chu kỳ và sau đó ở mức thấp tối thiểu một chu kỳ nữa để đảm bảo rằng sự chuyển trạng thái được phát hiện. IE0 và IE1 tự động được xóa khi CPU trỏ tới trình phục vụ ngắt tương ứng.

Nếu ngắt ngoài thuộc loại tác động mức, nguyên nhân ngắt ngoài phải được duy trì trạng thái tích cực cho đến khi ngắt theo yêu cầu thực sự được tạo ra. Sau đó nguyên nhân ngắt phải ở trạng thái thụ động trước khi trình phục vụ ngắt được thực thi xong hoặc trước khi có một ngắt khác được tạo ra.

Thông thường, một công việc được thực thi bên trong trình phục vụ ngắt làm cho nguyên nhân ngắt trả tín hiệu yêu cầu ngắt trở về trạng thái không tích cực.

**Thí dụ 6.4** : Điều khiển lò nung

*Sử dụng các ngắt để thiết kế bộ điều khiển lò nung sao cho nhiệt độ được duy trì ở $20^0C \pm 1^0C$.*




Hình 6.5 : Thí dụ (a) kết nối phần cứng (b) giản đồ thời gian

Giả sử ta dùng mạch giao tiếp sau đây cho thí dụ 6.4. Cuộn dây điều khiển mở / tắt lò được nối với chân P1.7 sao cho :

P1.7 = 1 : mở lò

P1.7 = 0 : tắt lò

Bộ cảm biến nhiệt độ được nối với $\overline{INT0}$, $\overline{INT1}$ và cung cấp các tín hiệu $\overline{HOT}$, $\overline{COLD}$ như sau :

$\overline{HOT} = 0$ nếu $T > 21^0C$ (P3.2)

$\overline{COLD} = 0$ nếu $T < 19^0C$ (P3.3)

Chương trình sẽ cho lò hoạt động ( mở lò ) khi $T < 19^0C$ và cho lò ngưng hoạt động ( tắt lò ) khi $T > 21^0C$.

Cấu hình phần cứng và giản đồ thời gian được trình bày ở hình 6.5.

```
          ORG    0
          LJMP   MAIN
                          ; vector ngắt ngoài 0 ở 0003H
EX0ISR :  CLR    P1.7     ; tắt lò
          RETI
          ORG    0013H
EX1ISR :  SETB   P1.7     ; mở lò
          RETI
          ORG    30H
MAIN :    MOV    IE, #85H ; cho phép các ngắt ngoài
          SETB   IT0      ; kích khởi cạnh âm
          SETB   IT1
          SETB   P1.7     ; mở lò
          JB     P3.2, SKIP ; nếu T > 21°C
          CLR    P1.7     ; tắt lò
SKIP :    SJMP   $        ; không làm gì.
          END
```

Ba lệnh đầu tiên trong chương trình chính cho phép các ngắt ngoài và xác định các ngắt thuộc loại tác động cạnh âm. Do trạng thái hiện tại của các ngõ vào $\overline{HOT}$ ( P3.2 ) và $\overline{COLD}$ ( P3.3 ) chưa được biết, 3 lệnh kế tiếp sẽ điều khiển mở hoặc tắt lò tùy trạng thái nhiệt độ hiện tại của lò. Trước tiên lò được mở ( SETB P1.7 ) và ngõ vào HOT được lấy mẫu ( JB P3.2, SKIP ). Nếu ngõ vào $\overline{HOT}$ ở mức cao, T < 21°C nên lệnh kế được bỏ qua và lò vẫn tiếp tục được mở.

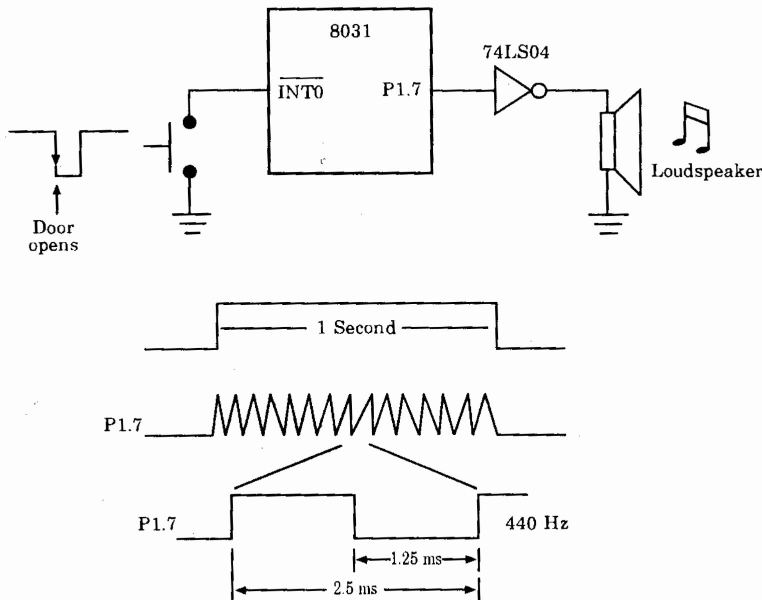
Ngược lại nếu ngõ vào HOT ở mức thấp, T > 21°C. Trong trường hợp này lệnh kế CLR P1.7 được thực thi để tắt lò trước khi đi vào vòng lặp " không làm gì ".

Lưu ý là phát biểu ORG 0003H không nhất thiết phải hiện diện ở ngay trước nhãn EX0ISR. Do lệnh LJMP MAIN dài 3 byte, EX0ISR chắc chắn được bắt đầu ở địa chỉ 0003H, điểm nhập của các ngắt ngoài 0.

**Thí dụ 6.5** : Hệ thống báo động

*Sử dụng các ngắt để thiết kế một hệ thống báo động tạo ra âm hiệu 400 Hz trong 1 sec ( sử dụng 1 loa nối với chân P1.7 ) mỗi khi bộ cảm biến đặt ở cửa ( được nối với chân* $\overline{INT0}$ *) tạo ra một chuyển trạng thái từ mức cao xuống mức thấp.*

Giải pháp cho thí dụ 6.5 là sử dụng 3 ngắt : ngắt ngoài 0 ( bộ cảm biến cửa ), ngắt do bộ định thời 0 ( âm hiệu 400 Hz ) và ngắt do bộ định thời 1 ( định thời 1 sec ). Cấu hình phần cứng và giản đồ thời gian được vẽ ở hình 6.6.




Hình 6.6 : Giao tiếp với loa sử dụng ngắt (a) kết nối phần cứng (b ) giản đồ thời gian

```
        ORG   0
        LJMP  MAIN          ; lệnh 3 byte
        LJMP  EX0ISR        ; địa chỉ vector EXT 0
        ORG   000BH         ; vector của bộ định thời 0
        LJMP  T0ISR
        ORG   001BH         ; vector của bộ định thời 1
        LJMP  T1ISR
        ORG   0030H
MAIN:   SETB  IT0           ; tác động cạnh âm.
```

```
          MOV   TMOD, #11H       ; chế độ định thời 16 bit
          MOV   IE, #81H         ; chỉ cho phép EX 0
          SJMP  $
EX0ISR :  MOV   R7, #20          ; 20x5000µs = 1 sec
          SETB  TF0              ; buộc ngắt do bộ định thời 0
          SETB  TF1              ; buộc ngắt do bộ định thời 1
          SETB  ET0              ; bắt đầu âm hiệu trong 1 sec
          SETB  ET1              ; cho phép các ngắt do bộ định thời
          RETI
T0ISR :   CLR   TR0              ; dừng bộ định thời.
          DJNZ  R7, SKIP         ; nếu chưa đủ 20 lần, thoát
          CLR   ET0              ; nếu đủ kết thúc âm hiệu
          CLR   ET1
          LJMP  EXIT
SKIP:     MOV   TH0, #HIGH(-50000)   ; trì hoãn 0,05sec
          MOV   TL0, #LOW(-50000)
          SETB  TR0
EXIT:     RETI
T1ISR:    CLR   TR1
          MOV   TH1, #HIGH(-1250)
          MOV   TL1, #LOW(-1250)
          CPL   P1.7
          SETB  TR1
          RETI
          END
```

Chương trình trên có 5 phần phân biệt : vị trí các vector ngắt, chương trình chính và ba trình phục vụ ngắt. Các vị trí vector ngắt chứa các lệnh LJMP để chuyển điều khiển đến các trình phục vụ ngắt tương ứng. Chương trình chính bắt đầu ở địa chỉ 0030H chỉ chứa 4 lệnh. Lệnh SETB IT0 cho phép ngõ vào ngắt ghép với bộ cảm biến cửa được kích khởi cạnh âm.

Lệnh MOV TMOD, #11H xác định chế độ hoạt động của cả hai bộ định thời là chế độ định thời 16-bit. Chỉ có ngắt ngoài 0 được phép bắt đầu ( lệnh MOV IE, #81H ) do điều kiện cửa mở là điều kiện cần phải có trước khi một ngắt nào đó được chấp nhận. Cuối cùng lệnh SJMP $ đặt chương trình chính vào vòng lặp " không làm gì ". Khi điều kiện cổng

mở được phát hiện ( bằng sự chuyển trạng thái từ mức cao xuống mức thấp ở chân $\overline{INT0}$ ), ngắt ngoài 0 được tạo ra.

Trình phục vụ cho ngắt ngoài 0 EX0ISR bắt đầu bằng việc nạp hằng số 20 cho R7 ( xem bên dưới ) rồi *set* cờ tràn của cả 2 bộ định thời bằng 1 để buộc các ngắt do bộ định thời xuất hiện. Tuy nhiên các ngắt do bộ định thời sẽ chỉ xuất hiện khi các bit tương ứng trong thanh ghi IE được cho phép. Hai lệnh kế tiếp SETB ET0 và SETB ET1 cho phép các ngắt do bộ định thời. Cuối cùng trình phục vụ ngắt ngoài 0 EX0ISR kết thúc bằng lệnh RETI để trở về chương trình chính.

Bộ định thời 0 tạo ra khoảng thời gian định thời 1 sec và bộ định thời 1 tạo ra âm hiệu 400 Hz. Sau khi trình phục vụ ngắt ngoài EX0ISR kết thúc, các ngắt do bộ định thời lập tức được tạo ra ( và được chấp nhận sau khi thực thi 1 lệnh SJMP $ ). Do chuỗi vòng cố định ( xem hình 6.2 ), ngắt do bộ định thời 0 được phục vụ trước tiên. Khoảng thời gian định thời 1 sec được tạo ra bằng cách lập trình để lặp lại 20 lần khoảng thời gian định thời 50000 μs. Thanh ghi R7 hoạt động như 1 bộ đếm.

Trình phục vụ ngắt T0ISR hoạt động như sau : trước tiên bộ định thời 0 được điều khiển ngưng và R7 được giảm bởi 1. Kế đến TH0/TL0 được nạp lại bởi giá trị –50.000, bộ định thời 0 được điều khiển hoạt động trở lại và ngắt được kết thúc. Ở lần ngắt thứ 20, R7 được giảm xuống 0 ( 1 sec đã trôi qua ). Các ngắt do cả 2 bộ định thời được vô hiệu ( CLR ET0, CLR ET1 ) và ngắt kết thúc. Không còn ngắt do bộ định thời được tạo ra nữa cho đến khi điều kiện cửa mở một lần nữa được phát hiện. Âm hiệu 400 Hz được lập trình bằng cách sử dụng các ngắt do bộ định thời 1. Tần số 400Hz yêu cầu chu kỳ là 2500 μs với 1250 μs ở mức cao và 1250 μs ở mức thấp. Trình phục vụ ngắt cho bộ định thời 1 chỉ đơn giản nạp –1250 cho TH1/TL1, lấy bù bit của *port* để kích loa và rồi kết thúc.

## 6.7 GIẢN ĐỒ THỜI GIAN CỦA NGẮT

Các ngắt được lấy mẫu và được chốt ở S5P2 của mỗi chu kỳ máy ( xem hình 6.7 ). Chúng được xoay vòng đến chu kỳ máy kế và nếu có một điều kiện ngắt tồn tại, ngắt được chấp nhận nếu :

a) không có ngắt nào khác có ưu tiên bằng hay cao hơn đang xảy ra.

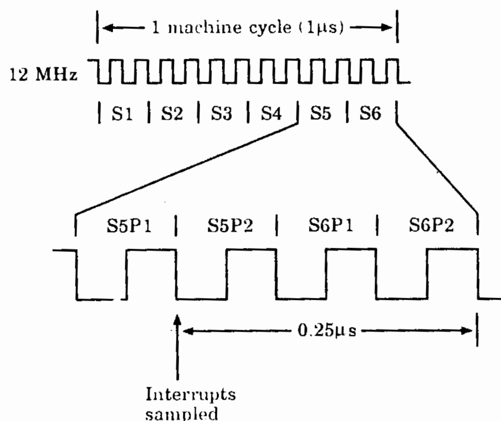
b) chu kỳ xoay vòng là chu kỳ cuối của một lệnh.

c) lệnh hiện hành không phải là lệnh RETI hoặc lệnh truy xuất đến thanh ghi IE hoặc IP. Trong suốt 2 chu kỳ kế tiếp, CPU cất nội

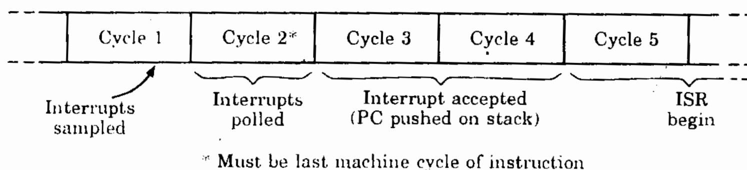
dung của PC vào *stack* và nạp cho PC địa chỉ vector ngắt. Trình phục vụ ngắt bắt đầu.

Điều kiện lệnh hiện hành không phải là lệnh RETI nhằm đảm bảo rằng có ít nhất một lệnh được thực thi sau mỗi một trình phục vụ ngắt.

Giản đồ thời gian được cho ở hình 6.8.




Hình 6.7 : Lấy mẫu các ngắt trong S5P2




Hình 6.8 : Xoay vòng các ngắt

Interrupts sampled : các ngắt được lấy mẫu

Machine cycle : chu kỳ máy

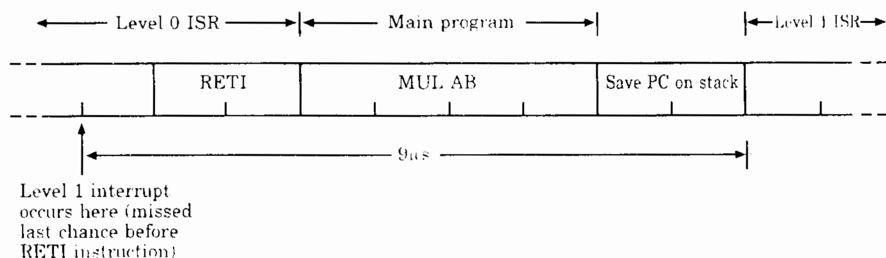
Interrupts polled : các ngắt được xoay vòng

Interrupt accepted ( PC pushed on stack ) : ngắt được chấp nhận ( PC được cất vào *stack* )

ISR begins : IRS bắt đầu

Thời gian từ lúc có một điều kiện ngắt xuất hiện đến khi trình phục vụ ngắt bắt đầu được gọi là *interrupt latency*. *Interrupt latency* rất quan trọng trong một số các ứng dụng điều khiển.

Với thạch anh 12 MHz, *interrupt latency* có thể ngắn khoảng 3.25 μs trên 8051. Một hệ thống sử dụng 8051 dùng ngắt ưu tiên cao sẽ có *interrupt latency* xấu nhất khoảng 9.25 μs ( giả sử ngắt ưu tiên cao luôn luôn được phép ). Điều này xảy ra nếu điều kiện ngắt xảy ra ngay trước khi có lệnh RETI của trình phục vụ ngắt mức 0 được theo sau bởi một lệnh nhân ( xem hình 6.9 ).




Hình 6.9 : *Interrupt latency*

Level 0 IRR : ISR mức 0

Main program : chương trình chính

Save PC on stack : cất PC vào *stack*

Level 1 interrupt occurs here ( missed last chance before RETI instruction ) : ngắt mức 1 xuất hiện ở đây ( cơ hội sau cùng bị bỏ qua trước khi có lệnh RETI )

# 7

# LẬP TRÌNH HỢP NGỮ

## 7.1 MỞ ĐẦU

Chương này giới thiệu về lập trình hợp ngữ ( assembly language programming ) trên *chip* vi điều khiển 8051. Hợp ngữ ( assembly language ) là ngôn ngữ của máy tính có vị trí ở giữa ngôn ngữ máy và ngôn ngữ cấp cao. Các ngôn ngữ cấp cao điển hình như Pascal và C sử dụng các từ và các phát biểu dễ hiểu đối với con người dù rằng còn khá xa mới đạt được mức độ dễ hiểu như ngôn ngữ tự nhiên. Ngôn ngữ máy ( machine language ) là ngôn ngữ ở dạng số nhị phân của máy tính. Một chương trình viết bằng ngôn ngữ máy là một chuỗi các byte nhị phân biểu diễn các lệnh mà máy tính thực thi được.

Hợp ngữ thay thế các mã nhị phân của ngôn ngữ máy bằng các mã gợi nhớ giúp ta dễ nhớ hơn và dễ lập trình hơn. Lấy thí dụ lệnh cộng trong ngôn ngữ máy được biễu diễn, chẳng hạn, bởi mã nhị phân là " 10110011 "; hợp ngữ thay thế bằng mã gợi nhớ " ADD ". Việc lập trình với các mã gợi nhớ rõ ràng được ưa chuộng hơn so với việc lập trình với các mã nhị phân. Dĩ nhiên công việc không phải dễ dàng như vừa trình bày. Các lệnh thao tác trên các dữ liệu và nơi chứa dữ liệu được chỉ ra bởi các kiểu định địa chỉ ( addressing mode ) khác nhau được bao gồm trong mã nhị phân của lệnh ngôn ngữ máy. Như vậy có thể có vài biến thể của lệnh ADD phụ thuộc vào " cái gì " được cộng. Các qui luật dùng để xác định các biến thể này là chủ đề trung tâm của việc lập trình hợp ngữ.

Một chương trình viết bằng hợp ngữ không thể được thực thi trực tiếp bởi máy tính. Sau khi được viết xong, chương trình này phải trải qua quá trình dịch thành ngôn ngữ máy. Trong thí dụ ở trên, mã gợi nhớ " ADD " phải được dịch thành mã nhị phân " 10110011 ". Phụ thuộc vào độ phức tạp của môi trường lập trình, việc dịch này có thể bao gồm một hoặc nhiều bước trước khi tạo ra sản phẩm là chương trình ngôn ngữ máy thực thi được. Ở mức tối thiểu, một chương trình được gọi là

trình dịch hợp ngữ ( assembler ) được cần đến để dịch các mã gợi nhớ của lệnh thành các mã nhị phân của ngôn ngữ máy. Một bước nữa có thể yêu cầu một trình liên kết ( linker ) để kết hợp các phần của chương trình ở các tập tin riêng rẽ và thiết lập địa chỉ trong bộ nhớ nơi mà chương trình được thực thi. Chúng ta sẽ bắt đầu với một vài định nghĩa.

Một chương trình viết bằng hợp ngữ ( gọi tắt là chương trình hợp ngữ ) là chương trình được viết dưới dạng các ký hiệu, các mã gợi nhớ, v.v... trong đó mỗi một phát biểu tương ứng với một lệnh của ngôn ngữ máy ( gọi tắt là lệnh ngôn ngữ máy hay lệnh máy ). Các chương trình hợp ngữ thường được gọi là chương trình nguồn hay mã nguồn ( hoặc mã ký hiệu ), chúng không được thực thi trực tiếp bởi máy tính. Một chương trình viết bằng ngôn ngữ máy ( gọi tắt là chương trình ngôn ngữ máy ) là chương trình chứa các mã nhị phân biểu diễn các lệnh của máy tính. Các chương trình ngôn ngữ máy thường được gọi là chương trình đối tượng hay mã đối tượng được thực thi bởi máy tính.

Trình dịch hợp ngữ là chương trình dùng để dịch một chương trình hợp ngữ thành chương trình ngôn ngữ máy. Chương trình ngôn ngữ máy ( mã đối tượng ) có thể ở dạng địa chỉ tuyệt đối ( absolute ) hoặc ở dạng tái định vị ( relocatable ). Trong dạng sau, việc liên kết được yêu cầu để thiết lập địa chỉ tuyệt đối từ đó chương trình được thực thi.

Trình liên kết là chương trình dùng để kết hợp các chương trình ( hay các mô-đun ) đối tượng ở dạng tái định vị và tạo ra một chương trình đối tượng ở dạng địa chỉ tuyệt đối để được thực thi bởi máy tính. Đôi khi người ta còn gọi trình liên kết là trình liên kết/định vị để phản ánh được hai chức năng riêng rẽ : chức năng kết hợp các mô-đun ở dạng tái định vị ( chức năng liên kết ), chức năng thiết lập các địa chỉ tuyệt đối để thực thi chương trình ( chức năng định vị ).

Một *segment* là một đơn vị của bộ nhớ chương trình hoặc bộ nhớ dữ liệu. Một *segment* có thể là tái định vị hay tuyệt đối. Một *segment* tái định vị có tên, loại ( kiểu : type ) và các thuộc tính ( attribute ) khác cho phép trình liên kết kết hợp *segment* này với các *segment* khác nếu cần, và để định vị chính xác *segment* này. Một *segment* tuyệt đối không có tên và không thể kết hợp với các *segment* khác.

Một mô-đun chứa một hoặc nhiều *segment*. Một mô-đun được gán một tên bởi người sử dụng. Các định nghĩa mô-đun xác định phạm vi của các ký hiệu cục bộ. Một tập tin đối tượng chứa một hoặc nhiều mô-đun. Một mô-đun có thể là một tập tin trong nhiều trường hợp cá biệt.

Một chương trình bao gồm một mô-đun tuyệt đối bằng cách kết hợp tất cả các *segment* tuyệt đối hay tái định vị từ tất cả các mô-đun tạo

thành chương trình. Một chương trình chỉ chứa các mã nhị phân của các lệnh ( với các địa chỉ và các hằng dữ liệu ) và máy tính hiểu các mã nhị phân này.

## 7.2 TRÌNH DỊCH HỢP NGỮ

Có nhiều trình dịch hợp ngữ và các chương trình hỗ trợ khác cho phép ta dễ dàng phát triển các ứng dụng trên *chip* vi điều khiển 8051. Trình dịch hợp ngữ họ MCS-51 của Intel ( ASM51 ) được dùng làm chuẩn để so sánh với các trình dịch hợp ngữ khác. Trong chương này chúng ta tập trung nghiên cứu việc lập trình hợp ngữ bằng cách sử dụng hầu hết các đặc trưng của MCS-51.

Mặc dù có nhiều đặc trưng đã được chuẩn hóa, có thể có vài đặc trưng không được hiện thực trong các trình dịch hợp ngữ của các công ty khác.

ASM51 là trình dịch hợp ngữ mạnh, hoạt động tốt trên các hệ thống của Intel và trên các họ máy tính của IBM-PC. Do các máy tính này chứa các *chip* khác với 8051, ASM51 được gọi là trình dịch hợp ngữ chéo. Chương trình nguồn viết cho 8051 có thể được soạn thảo trên máy tính ( bằng một phần mềm soạn văn bản ) và có thể được hợp dịch thành một tập tin đối tượng và một tập tin liệt kê ( listing file ) bằng ASM51. Chương trình đối tượng này không thể thực thi được trên máy tính. Do *chip* CPU của máy tính không phải là *chip* 8051, các mã nhị phân trong tập tin đối tượng không thể được hiểu bởi máy tính. Việc thực thi chương trình vừa nêu trên máy tính yêu cầu phải có hoặc phần cứng tương thích hoặc phần mềm mô phỏng 8051. Khả năng thứ ba là nạp chương trình đối tượng vào một hệ thống có sử dụng 8051 để thực thi. Phần cứng tương thích, phần mềm mô phỏng và việc nạp chương trình đối tượng vào một hệ thống có sử dụng 8051 cùng với các kỹ thuật khác thường được đề cập trong nhiều tài liệu khác nhau.

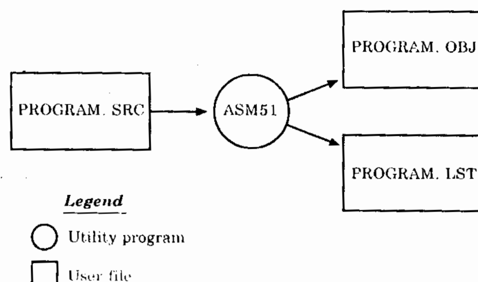
ASM51 được gọi từ dấu nhắc hệ thống bằng lệnh :

**ASM51 source_file [assembler_controls]**

Tập tin nguồn ( source_file ) được hợp dịch và các điều khiển của trình dịch hợp ngữ ( assembler_controls ) tạo ra hiệu quả. Các điều khiển của trình dịch hợp ngữ là các tùy chọn sẽ được đề cập sau trong chương này. Trình dịch hợp ngữ nhận tập tin nguồn ( thí dụ PROGRAM.SRC ) và tạo ra tập tin đối tượng ( PROGRAM.OBJ ), tập tin liệt kê ( PROGRAM.LST ). Điều này được minh họa trong hình 7.1.

Do hầu hết các trình dịch hợp ngữ quét tập tin nguồn hai lần trong quá trình dịch tập tin nguồn sang ngôn ngữ máy, chúng còn được gọi là trình dịch hợp ngữ hai bước. Trình dịch hợp ngữ sử dụng một bộ đếm vị

trí LC ( location counter ) để xác định địa chỉ của các lệnh và các nhãn. Hoạt động của mỗi bước được mô tả sau đây.




Hình 7.1 : Hợp dịch một chương trình nguồn

Utility program : chương trình tiện ích

User file : tập tin của người sử dụng

### 7.2.1 Bước 1

Trong bước 1, tập tin nguồn được quét từng dòng một và một bảng ký hiệu ( symbol table ) được tạo ra. Bộ đếm vị trí được mặc định bằng 0 hoặc được *set* giá trị ban đầu bởi chỉ dẫn ORG. Khi tập tin được quét, bộ đếm vị trí được tăng bởi kích thước của mỗi một lệnh. Các chỉ dẫn định nghĩa dữ liệu DB hoặc DW tăng bộ đếm vị trí bởi số byte được định nghĩa. Các chỉ dẫn dự trữ ( hay dành trước ) bộ nhớ DS ( reserve memory directive ) tăng bộ đếm vị trí bởi số byte được dự trữ.

Mỗi một lần tìm thấy một nhãn ở trước một dòng lệnh, nhãn được đặt vào trong bảng ký hiệu cùng với giá trị hiện hành của bộ đếm vị trí. Các ký hiệu được định nghĩa bởi các chỉ dẫn gán ( equate directive ) như EQU được đặt vào trong bảng ký hiệu cùng với giá trị được gán. Bảng ký hiệu được lưu lại và sau đó được sử dụng cho bước 2.

### 7.2.2 Bước 2

Trong bước 2 các tập tin đối tượng và tập tin liệt kê được tạo ra. Các mã gợi nhớ được chuyển đổi thành các opcode và được đưa vào các tập tin trên. Các toán hạng được đánh giá và được đặt sau opcode của lệnh. Ở nơi các ký hiệu xuất hiện trong trường toán hạng, các giá trị của chúng được lấy ra từ bảng ký hiệu ( đã được tạo ra trong bước 1 ) và được dùng để tính toán dữ liệu hoặc địa chỉ cho các lệnh.

Do việc hợp dịch được tiến hành theo 2 bước, chương trình nguồn có thể sử dụng " tham chiếu thuận ", nghĩa là có thể sử dụng một ký hiệu trước khi ký hiệu này được định nghĩa. Thí dụ khi có điều khiển rẽ

nhánh theo chiều tăng của địa chỉ đến một nhãn chưa được định nghĩa trong một chương trình.

Tập tin đối tượng, nếu thuộc dạng địa chỉ tuyệt đối, chỉ chứa các byte nhị phân ( 00H – FFH ) của chương trình ngôn ngữ máy. Tập tin đối tượng thuộc loại tái định vị cũng sẽ chứa một bảng ký hiệu và các thông tin khác cần cho sự liên kết và định vị sau này.

Tập tin liệt kê chứa các mã văn bản ASCII ( 20H – 7EH ) cho chương trình nguồn và các byte số hex trong chương trình mã máy.

Một minh họa tốt để phân biệt tập tin đối tượng và tập tin liệt kê là cho hiển thị chúng trên màn hình của máy tính. Tập tin liệt kê được hiển thị một cách rõ ràng với mỗi một dòng chứa địa chỉ, opcode và có thể dữ liệu, tiếp theo là phát biểu của chương trình từ tập tin nguồn. Điều này cũng dễ hiểu vì tập tin liệt kê chỉ chứa các mã ASCII văn bản. Ngược lại tập tin đối tượng do chứa các mã nhị phân của chương trình ngôn ngữ máy của 8051 nên được hiển thị như là các " rác " và ta không thể đọc được.

## 7.3 KHUÔN DẠNG CỦA CHƯƠNG TRÌNH HỢP NGỮ

Các chương trình hợp ngữ chứa :

- các lệnh ( instruction ) của bộ vi xử lý, bộ vi điều khiển
- các chỉ dẫn ( directive ) của trình dịch hợp ngữ
- các điều khiển ( control ) của trình dịch hợp ngữ
- các chú thích ( comment )

Các lệnh là các mã gợi nhớ quen thuộc của các lệnh thực thi được ( như là ANL ). Các chỉ dẫn của trình dịch hợp ngữ là các lệnh của trình dịch hợp ngữ dùng để định nghĩa cấu trúc chương trình, các ký hiệu, dữ liệu, các hằng số và v.v... ( như là ORG ). Các điều khiển của trình dịch hợp ngữ thiết lập các chế độ của trình dịch hợp ngữ và các luồng hợp dịch trực tiếp ( direct assembly flow ) ( như là $TITLE ). Các chú thích giúp cho chương trình dễ đọc bằng cách đưa ra các giải thích về mục đích và hoạt động của các chuỗi lệnh.

Các dòng chứa các lệnh và các chỉ dẫn phải được viết theo các qui luật mà trình dịch hợp ngữ hiểu được. Mỗi một dòng được chia thành các trường cách biệt nhau bởi khoảng trắng hoặc khoảng tab. Khuôn dạng tổng quát của mỗi một dòng như sau :

**[label:]  mnemonic  [operand][, operand][ . . . ] [; comment]**

label : nhãn

mnemonic : mã gợi nhớ

operand : toán hạng

comment : chú thích

Chỉ có trường mã gợi nhớ là bắt buộc. Nhiều trình dịch hợp ngữ yêu cầu trường nhãn, nếu hiện diện, phải bắt đầu từ bên trái ở cột 1, và các trường tiếp theo được cách nhau bởi ký tự khoảng trắng hoặc tab. Với ASM51, trường nhãn không cần phải bắt đầu ở cột 1 và trường mã gợi nhớ không cần ở trên cùng một dòng với trường nhãn. Tuy nhiên trường toán hạng phải ở trên cùng một dòng với trường mã gợi nhớ. Các trường được mô tả như sau.

### 7.3.1 Trường nhãn (label)

Nhãn biểu thị địa chỉ của lệnh ( hoặc dữ liệu ) theo sau. Khi có sự rẽ nhánh đến lệnh này, nhãn được dùng trong trường toán hạng của lệnh rẽ nhánh hoặc nhảy ( như là SJMP SKIP ).

Trong khi thuật ngữ " nhãn " luôn luôn biểu thị một địa chỉ, thuật ngữ " ký hiệu " lại tổng quát hơn. Nhãn là một loại ký hiệu và được nhận dạng bằng dấu : ( kết thúc nhãn ).

Các ký hiệu được gán giá trị hoặc thuộc tính bằng cách sử dụng các chỉ dẫn như là EQU, SEGMENT, BIT, DATA, v.v... Các ký hiệu có thể là các địa chỉ, các hằng dữ liệu, tên của các *segment* hoặc các cấu trúc khác do người lập trình nghĩ ra. Các ký hiệu không kết thúc bằng dấu :. Trong thí dụ sau đây PAR là một ký hiệu còn START là một nhãn :

```
PAR       EQU    500
START:    MOV    A, #0FFH
```

Một ký hiệu ( hoặc nhãn ) phải bắt đầu bằng một ký tự chữ hoặc dấu hỏi ( ? ) hoặc dấu nối dưới ( _ ) và tiếp theo phải là các ký tự chữ, các số, dấu " ? " hoặc " _ "; ký hiệu ( hoặc nhãn ) có thể dài 31 ký tự ở dạng chữ thường hoặc chữ in. Các ký hiệu ( hoặc nhãn ) không thể trùng với các từ khóa ( các mã gợi nhớ, các chỉ dẫn, các toán tử hoặc các ký hiệu tiền định nghĩa ).

### 7.3.2 Trường mã gợi nhớ √ (mnemonic)

Mã gợi nhớ của lệnh hoặc chỉ dẫn của trình dịch hợp ngữ theo sau trường nhãn. Các thí dụ về mã gợi nhớ của lệnh là ADD, MOV, DIV hoặc INC. Các thí dụ về mã gợi nhớ của chỉ dẫn là ORG, EQU hoặc DB. Các chỉ dẫn được mô tả sau trong chương này.

### 7.3.3 Trường toán hạng

Trường toán hạng theo sau trường mã gợi nhớ. Trường này chứa địa chỉ hoặc dữ liệu mà lệnh sẽ sử dụng. Một nhãn có thể được dùng để biểu thị địa chỉ của dữ liệu hoặc một ký hiệu có thể được dùng để biểu thị một hằng dữ liệu. Các khả năng của trường toán hạng phụ thuộc vào thao tác. Có thao tác không có toán hạng ( thí dụ lệnh RET ) trong khi các thao tác khác cho phép nhiều toán hạng cách nhau bởi dấu phẩy. Do vậy có nhiều khả năng cho trường toán hạng mà chúng ta sẽ lần lượt khảo sát sau.

### 7.3.4 Trường chú thích

Các ghi chú để làm rõ chương trình được đặt trong trường chú thích ở cuối dòng lệnh. Các chú thích cần phải được bắt đầu bằng dấu chấm phẩy ( ; ). Các chú thích có thể chiếm nhiều dòng riêng và cũng phải bắt đầu bằng dấu chấm phẩy ( ; ). Các chương trình con và các phần có kích thước lớn của chương trình thường bắt đầu bởi một khối chú thích bao gồm nhiều dòng chú thích để giải thích các đặc trưng tổng quát của chương trình con hoặc phần mềm theo sau.

### 7.3.5 Các ký hiệu đặc biệt

Các ký hiệu đặc biệt được dùng cho các kiểu định địa chỉ thanh ghi. Các ký hiệu này bao gồm A, Ro đến R7, DPTR, PC, C và AB. Dấu $ cũng là một ký hiệu đặc biệt được dùng để tham chiếu đến giá trị hiện hành của bộ đếm vị trí. Một số thí dụ như sau :

```
SETB   C
INC    DPTR
JNB    TI, $
```

Lệnh sau cùng sử dụng bộ đếm vị trí của ASM51 để tránh không dùng một nhãn. Lệnh trên có thể viết lại như sau :

```
HERE:   JNB   TI, HERE
```

### 7.3.6 Địa chỉ gián tiếp

Với một số lệnh, trường toán hạng có thể xác định một thanh ghi mà nội dung của thanh ghi là địa chỉ của dữ liệu. Dấu @ chỉ ra một địa chỉ gián tiếp và thanh ghi theo sau chỉ có thể là R0, R1, DPTR hoặc PC tùy vào lệnh cụ thể. Thí dụ :

```
ADD    A, @R0
MOVC   A, @A+PC
```

Lệnh đầu tiên lấy ra một byte dữ liệu từ RAM nội ở địa chỉ là nội dung của thanh ghi R0. Lệnh thứ hai lấy ra một byte dữ liệu từ bộ nhớ chương trình ngoài ở địa chỉ là nội dung của thanh chứa A cộng với nội dung của bộ đếm chương trình PC. Lưu ý là nội dung của bộ đếm chương trình PC khi cộng là địa chỉ của lệnh tiếp theo lệnh MOVC. Với cả hai lệnh trên, giá trị lấy ra được đưa vào cất trong thanh chứa A.

### 7.3.7 Dữ liệu tức thời

Các lệnh sử dụng kiểu định địa chỉ tức thời cung cấp dữ liệu trong trường toán hạng và dữ liệu này trở thành một phần của lệnh. Dữ liệu tức thời được đứng trước bởi dấu #. Thí dụ :

```
CONSTANT        EQU     100
                MOV     A, #0FEH
                ORL     40H, #CONSTANT
```

Các lệnh trên dữ liệu tức thời ( ngoại trừ lệnh MOV DPTR, #data ) đều yêu cầu dữ liệu 8-bit. Dữ liệu tức thời được đánh giá như là một hằng số 16-bit và sau đó byte thấp được sử dụng. Tất cả các bit trong byte cao phải giống nhau ( 00H hoặc FFH ) hoặc một thông báo lỗi " giá trị sẽ không lấp đầy một byte " được tạo ra. Thí dụ các lệnh sau đây hợp lệ :

```
MOV     A, #0FF00H
MOV     A, #00FFH
```

còn các lệnh sau sẽ tạo ra thông báo lỗi :

```
MOV     A, #0FE00H
MOV     A, #01FFH
```

Nếu sử dụng số dạng thập phân có dấu, các hằng số từ – 256 đến + 256 có thể được sử dụng. Thí dụ hai lệnh sau tương đương và hợp lệ :

```
MOV     A, # -256
MOV     A, #0FF00H
```

cả hai lệnh trên đều đặt 00H vào thanh chứa A.

### 7.3.8 Địa chỉ dữ liệu

Nhiều lệnh truy xuất các vị trí nhớ bằng cách sử dụng kiểu định địa chỉ trực tiếp và yêu cầu một địa chỉ của bộ nhớ dữ liệu trên *chip* ( 00H – 7FH ) hoặc địa chỉ của một thanh ghi chứa năng đặc biệt SFR ( 80H – 0FFH ) trong trường toán hạng. Các ký hiệu tiền định nghĩa có thể được

sử dụng thay cho địa chỉ của các thanh ghi chức năng đặc biệt SFR. Thí dụ :

```
MOV   A, 45H
MOV   A, SBUF       ; tương đương lệnh MOV  A, 99H
```

### 7.3.9 Địa chỉ bit

Một trong hầu hết các đặc trưng mạnh của 8051 là khả năng truy xuất các bit riêng rẽ mà không cần thao tác lập mặt nạ trên các byte. Các lệnh truy xuất các vị trí được định địa chỉ bit phải cung cấp địa chỉ bit trong bộ nhớ dữ liệu nội ( 00H – 7FH ) hoặc địa chỉ bit trong các thanh ghi SFR ( 80H – 0FFH ).

Ta có ba cách để xác định địa chỉ bit trong một lệnh :

(a) địa chỉ bit đã biết trước

(b) sử dụng toán tử dot ( . ) giữa địa chỉ byte và vị trí bit

(c) sử dụng ký hiệu tiền định nghĩa. Dưới đây là các thí dụ :

```
SETB   0E7H      ; địa chỉ bit biết trước
SETB   ACC.7     ; sử dụng toán tử dot ( . )
JNB    TI, $     ; TI là ký hiệu tiền định nghĩa
JNB    99H, $    ; cùng công dụng với lệnh trên
```

### 7.3.10 Địa chỉ của lệnh

Một địa chỉ của lệnh được dùng trong trường toán hạng cho các lệnh nhảy, bao gồm các lệnh nhảy tương đối ( SJMP và các lệnh nhảy có điều kiện ), các lệnh nhảy và gọi tuyệt đối ( ACALL, AJMP ) và các lệnh nhảy và gọi dài ( LJMP, LCALL ).

Địa chỉ của lệnh thường được cho dưới dạng các nhãn. Thí dụ :

```
HERE :  .
        .
        .
        SJMP   HERE
```

ASM51 sẽ xác định địa chỉ đúng của lệnh và chèn vào lệnh rẽ nhánh hoặc *offset* có dấu 8-bit hoặc địa chỉ trang 11-bit hoặc địa chỉ dài 16-bit một cách thích hợp.

### 7.3.11 Các lệnh nhảy và gọi tổng quát

ASM51 cho phép người lập trình sử dụng mã gợi nhớ tổng quát CALL hoặc JMP. " JMP " có thể được sử dụng thay cho SJMP, AJMP hoặc LJMP và " CALL " có thể được sử dụng thay cho ACALL và LCALL. Trình dịch hợp ngữ biến đổi mã gợi nhớ tổng quát thành lệnh " thực " theo một vài qui luật đơn giản. Mã gợi nhớ tổng quát biến đổi thành dạng ngắn ( chỉ với JMP ) nếu không có tham chiếu thuận được sử dụng và đích nhảy đến ở trong khoảng −128 byte hoặc thành dạng tuyệt đối nếu không có tham chiếu thuận được sử dụng và lệnh tiếp theo lệnh JMP hoặc CALL ở trong cùng khối 2 KB với lệnh ở đích. Nếu các dạng ngắn và tuyệt đối không được sử dụng, dạng dài sẽ được sử dụng khi biến đổi.

| LOC | OBJ | LINE | SOURCE | | |
|---|---|---|---|---|---|
| 1234 | | 1 | | ORG | 1234H |
| 1234 | 04 | 2 | START: | INC | A |
| 1235 | 80F0 | 3 | | JMP | START ; SJMP |
| 12FC | | 4 | | ORG | START + 200 |
| 12FC | 4134 | 5 | | JMP | START |
| 12FE | 021301 | 6 | | JMP | FINISH |
| 1301 | 04 | 7 | FINISH: | INC | A |
| | | 8 | | END | |

Hình 7.2 : Sử dụng mã gợi nhớ JMP tổng quát

LOC : vị trí
OBJ : mã đối tượng
LINE : dòng
SOURCE : mã nguồn

Việc biến đổi không nhất thiết phải có sự lựa chọn lập trình tốt nhất. Thí dụ rẽ nhánh nếu là thuận và cách một vài lệnh, dạng tổng quát JMP sẽ luôn luôn biến đổi thành dạng dài LJMP ngay cả khi nếu thay bằng SJMP có lẽ sẽ tốt hơn. Ta hãy khảo sát một chuỗi lệnh đã được hợp dịch ở hình 7.2 với 3 lệnh nhảy tổng quát được sử dụng.

Lệnh nhảy thứ nhất ( dòng 3 ) được hợp dịch như lệnh SJMP do bởi đích nhảy đến ở trước lệnh nhảy ( nghĩa là không có tham chiếu thuận ) và *offset* nhỏ hơn − 128. Chỉ dẫn ORG trong dòng 4 tạo ra một khoảng cách 200 vị trí giữa nhãn START và lệnh nhảy thứ hai, lệnh nhảy ở dòng 5 được biến đổi thành AJMP do bởi *offset* bây giờ lớn quá tầm của

SJMP. Cũng cần lưu ý là địa chỉ tiếp theo lệnh nhảy thứ hai ( 12FCH ) và địa chỉ của START ( 1234H ) ở trong cùng một trang 2 KB ( từ 1000H đến 17FFH đối với chuỗi lệnh trong thí dụ ). Đây là tiêu chuẩn phải được thỏa cho địa chỉ tuyệt đối. Lệnh nhảy thứ ba được hợp dịch như lệnh LJMP do bởi đích nhảy ( FINISH ) chưa được định nghĩa khi lệnh nhảy được hợp dịch ( nghĩa là một tham chiếu thuận được sử dụng ).

## 7.4 ĐÁNH GIÁ BIỂU THỨC TRONG THỜI GIAN DỊCH

Các giá trị và hằng số trong trường toán hạng có thể được biểu diễn theo ba cách :

(a) một cách tường minh ( thí dụ 0EFH )

(b) dùng ký hiệu tiền định nghĩa ( thí dụ ACC )

(c) dùng một biểu thức ( thí dụ 2 + 3 ).

Việc sử dụng các biểu thức cho ta một kỹ thuật mạnh để làm cho các chương trình hợp ngữ dễ đọc hơn và linh hoạt hơn. Khi một biểu thức được sử dụng, trình dịch hợp ngữ tính toán giá trị và chèn kết quả vào trong lệnh.

Mọi việc tính toán biểu thức được thực hiện bằng cách sử dụng số 16-bit; tuy nhiên hoặc 8-bit hoặc 16-bit được chèn vào trong lệnh tùy nhu cầu. Thí dụ hai lệnh sau đây giống nhau :

```
MOV    DPTR, #04FFH + 3
MOV    DPTR, #0502H            ; kết quả 16-bit được sử dụng
```

Tuy nhiên nếu biểu thức trên được sử dụng cho lệnh " MOV A, #data ", thông báo lỗi " giá trị sẽ không lấp đầy trong byte " được tạo ra bởi ASM51. Một cách tổng quát, các qui luật cho việc đánh giá biểu thức như sau.

### 7.4.1 Cơ số

Cơ số cho các hằng số được chỉ ra theo cách thông dụng của các bộ vi xử lý của Intel. Các hằng số phải được theo sau bởi " B " cho số nhị phân, " O " hoặc " Q " cho số octal, " D " hoặc không có cho số thập phân và " H " cho số hex. Thí dụ các lệnh sau đây giống nhau :

```
MOV    A, #15
MOV    A, #1111B
MOV    A, #0FH
MOV    A, #17Q
MOV    A, #15D
```

Lưu ý là một digit số phải là ký tự đầu tiên cho các hằng số dạng số hex để phân biệt chúng với các ký tự ( thí dụ "0A5H" thay vì là "A5H" ).

### 7.4.2 Các chuỗi ký tự

Các chuỗi có một hoặc hai ký tự có thể được sử dụng làm các toán hạng trong các biểu thức. Các mã ASCII được biến đổi thành số nhị phân tương đương bởi trình dịch hợp ngữ. Các hằng ký tự được đặt trong 2 dấu nháy đơn.

Sau đây là một vài thí dụ :

```
CJNE   A, # 'Q', AGAIN
SUBB   A, # '0'
MOV    DPTR, # 'AB'
MOV    DPTR, #4142H
```

### 7.4.3 Các toán tử số học

Các toán tử số học là :

| | |
|---|---|
| + | cộng |
| – | trừ |
| * | nhân |
| / | chia |
| MOD | modulo |

Thí dụ, hai lệnh sau tương đương :

```
MOV    A, #10 + 10H
MOV    A, #1AH
```

Hai lệnh sau cũng tương đương :

```
MOV    A, #25 MOD 7
MOV    A, #4
```

Vì toán tử MOD có thể bị nhầm lẫn với một ký hiệu, toán tử này phải cách các toán hạng tối thiểu một ký tự khoảng trắng hoặc tab, hoặc các toán hạng phải được ở trong hai dấu ngoặc. Cũng áp dụng tương tự cho các toán hạng khác bao gồm các ký tự chữ.

### 7.4.4 Các toán tử logic

Các toán tử logic là : OR, AND, XOR và NOT.

Các thao tác được thực hiện trên các bit tương ứng trong từng toán hạng. Các toán tử phải được cách các toán hạng bởi ký tự khoảng trắng hay tab.

Thí dụ hai lệnh sau tương đương :

```
MOV   A, # '9' AND 0FH
MOV   A, #9
```

Toán tử NOT chỉ thực hiện trên một toán hạng. Ba lệnh MOV sau giống nhau :

```
THREE          EQU   3
MINUS_THREE    EQU   - 3
               MOV   A, # (NOT THREE) + 1
               MOV   A, #MINUS_THREE
               MOV   A, #11111101B
```

### 7.4.5 Các toán tử đặc biệt

Các toán tử đặc biệt là :

| | |
|---|---|
| SHR | dịch phải |
| SHL | dịch trái |
| HIGH | byte cao |
| LOW | byte thấp |
| ( ) | được đánh giá trước |

Thí dụ hai lệnh sau tương đương :

```
MOV   A, #8 SHL 1
MOV   A, 10H
```

Hai lệnh sau cũng tương đương :

```
MOV   A, #HIGH 1234H
MOV   A, #12H
```

### 7.4.6 Các toán tử quan hệ

Khi một toán tử quan hệ được sử dụng giữa hai toán hạng, kết quả luôn luôn là sai ( false : 0000H ) hoặc đúng ( true : FFFFH ).

Các toán tử quan hệ là :

| | | |
|---|---|---|
| EQ | = | bằng |

| | | |
|---|---|---|
| NE | <> | không bằng |
| LT | < | nhỏ hơn |
| LE | <= | nhỏ hơn hoặc bằng |
| GT | > | lớn hơn |
| GE | >= | lớn hơn hoặc bằng |

Với mỗi một toán tử ta có thể sử dụng một trong hai dạng ký hiệu nêu trên ( thí dụ EQ hoặc = ). Trong các thí dụ sau các quan hệ đều cho kết quả đúng ( true ) :

```
MOV     A, #5 = 5
MOV     A, #5 NE 4
MOV     A, # 'X' LT 'Z'
MOV     A, # 'X' >= 'X'
MOV     A, # $ > 0
MOV     A, #100 GE 50
```

do các lệnh trên sau khi được dịch đều trở thành

```
MOV     A, #0FFH
```

Mặc dù các biểu thức được đánh giá để cho kết quả 16-bit ( nghĩa là 0FFFFH ), trong các thí dụ trên chỉ có byte thấp sử dụng vì lệnh thực hiện thao tác di chuyển byte ( và ta lưu ý là các số có dấu FFFFH và FFH có giá trị bằng nhau và bằng – 1 ).

### 7.4.7 Các thí dụ cho biểu thức

Dưới đây là các thí dụ cho biểu thức và các kết quả :

| **Biểu thức** | **Kết quả** |
|---|---|
| 'B' – 'A' | 0001H |
| 8/3 | 0002H |
| 155 MOD 2 | 0001H |
| 4 * 4 | 0010H |
| 8 AND 7 | 0000H |
| NOT 1 | FFFEH |
| 'A' SHL 8 | 4100H |
| LOW 65535 | 00FFH |
| (8 + 1) * 2 | 0012H |

| | |
|---|---|
| 5 EQ 4 | 0000H |
| 'A' LT 'B' | FFFFH |
| 3 <= 3 | FFFFH |

Một thí dụ thực tế minh họa một thao tác chung cho việc khởi động bộ định thời như sau : đặt –500 vào các thanh ghi TH1 và TL1 của bộ định thời 1.

Bằng cách sử dụng các toán tử HIGH và LOW, một phương pháp tốt là :

```
VALUE       EQU   -500
            MOV   TH1, #HIGH  VALUE
            MOV   TL1, #LOW  VALUE
```

Trình dịch hợp ngữ biến đổi –500 thành giá trị 16-bit tương ứng là FE0CH, sau đó các toán tử HIGH và LOW sẽ tách byte cao FEH và byte thấp 0CH để các lệnh MOV nạp cho TH1 và TL1.

### 7.4.8 Ưu tiên của các toán tử

Ưu tiên của các toán tử trong biểu thức từ cao nhất đến thấp nhất như sau :

( )

HIGH LOW

* / MOD SHL SHR

\+ –

EQ NE LT LE GT GE = <> <= >=

NOT

AND

OR XOR

Khi các toán tử có cùng ưu tiên được sử dụng, chúng được đánh giá theo thứ tự từ trái sang phải. Thí dụ :

| **Biểu thức** | **Giá trị** |
|---|---|
| HIGH ( 'A' SHL 8 ) | 0041H |
| HIGH 'A' SHL 8 | 0000H |
| NOT 'A' – 1 | FFBFH |
| 'A' OR 'A' SHL 8 | 4141H |

## 7.5 CÁC CHỈ DẪN

Các chỉ dẫn là các lệnh đối với trình dịch hợp ngữ. Các chỉ dẫn không được hợp dịch và không phải là các lệnh của hợp ngữ để được thực thi bởi bộ vi xử lý. Tuy nhiên các chỉ dẫn lại được đặt trong trường mã gợi nhớ của chương trình. Ngoài các ngoại lệ là DB và DW, các chỉ dẫn không có ảnh hưởng trực tiếp lên nội dung của bộ nhớ.

ASM51 cung cấp cho ta một vài loại chỉ dẫn sau :

- Điều khiển trạng thái của trình dịch hợp ngữ ( ORG, END, USING )
- Định nghĩa ký hiệu ( SEGMENT, EQU, SET, DATA, IDATA, XDATA, BIT, CODE )
- Dành trước vùng nhớ / khởi động vùng nhớ (DS, DBIT, DB, DW)
- Liên kết chương trình ( PUBLIC, EXTRN, NAME )
- Lựa chọn *segment* ( RSEG, CSEG, DSEG, ISEG, BSEG, XSEG )

### 7.5.1 Điều khiển trạng thái

#### 7.5.1.1 ORG ( set origin )

Dạng của chỉ dẫn ORG như sau :

```
ORG    expression
```

Expression : biểu thức

Chỉ dẫn ORG thay đổi nội dung bộ đếm vị trí để thiết lập một gốc mới của chương trình cho các phát biểu theo sau. Nhãn không được phép sử dụng. Thí dụ :

```
ORG    100H                        ; bộ đếm vị trí được thiết
                                   ; lập bằng 100H
ORG    ($ + 1000H) AND 0F000H      ; thiết lập đến 4 K kế
```

Chỉ dẫn ORG có thể sử dụng trong bất kỳ loại *segment* nào. Nếu *segment* hiện hành là tuyệt đối, giá trị sẽ là địa chỉ tuyệt đối trong *segment* hiện hành. Nếu một *segment* tái định vị được tích cực, giá trị của biểu thức được xử lý như là một *offset* từ địa chỉ nền của thể hiện ( instance ) hiện hành của *segment*.

#### 7.5.1.2 END

Dạng của chỉ dẫn END như sau :

```
END
```

END nên là phát biểu cuối cùng của chương trình nguồn. Nhãn không được phép sử dụng và không có gì theo sau phát biểu END được xử lý bởi trình dịch hợp ngữ.

#### 7.5.1.3 USING

Dạng của chỉ dẫn USING như sau :

```
USING        expression
```

Expression : biểu thức

Chỉ dẫn này thông báo cho ASM51 về dãy thanh ghi tích cực hiện hành. Sau phát biểu dùng chỉ dẫn USING, các ký hiệu tiền định từ AR0 đến AR7 được sử dụng thay cho các ký hiệu thanh ghi từ R0 đến R7 và chúng sẽ được biến đổi thành địa chỉ trực tiếp tương ứng trong dãy thanh ghi tích cực. Ta hãy khảo sát chuỗi lệnh sau :

```
USING        3
PUSH         AR7
USING        1
PUSH         AR7
```

lệnh PUSH đầu tiên trong thí dụ trên được dịch thành PUSH 1FH ( R7 trong dãy thanh ghi 3 ) trong khi lệnh PUSH thứ hai được dịch thành PUSH 0FH ( R7 trong dãy thanh ghi 1 ).

Lưu ý là chỉ dẫn USING thực tế không chuyển đổi các dãy thanh ghi; chỉ dẫn này chỉ thông báo cho ASM51 về dãy thanh ghi tích cực. Việc thực thi các lệnh của 8051 đòi hỏi phải chuyển đổi các dãy thanh ghi. Điều này được minh họa bằng việc sửa đổi thí dụ trên như sau :

```
MOV          PSW, #00011000B       ; chọn dãy thanh ghi 3
USING        3
PUSH         AR7                   ; dịch thành PUSH 1FH
MOV          PSW, #00001000B       ; chọn dãy thanh ghi 1
USING        1
PUSH         AR7                   ; dịch thành PUSH 0FH
```

### 7.5.2 Định nghĩa ký hiệu

Các chỉ dẫn định nghĩa ký hiệu tạo ra các ký hiệu biểu diễn các *segment*, các thanh ghi, các số và các địa chỉ. Các chỉ dẫn này không được có nhãn đứng trước. Các ký hiệu được định nghĩa bằng các chỉ dẫn định nghĩa ký hiệu phải không được định nghĩa trước và phải không

được định nghĩa sau đó bằng bất cứ phương tiện nào. Chỉ dẫn SET là ngoại lệ duy nhất. Các chỉ dẫn định nghĩa ký hiệu được đề cập dưới đây.

#### 7.5.2.1 *Segment*

Dạng của chỉ dẫn SEGMENT như sau :

symbol SEGMENT segment_type

symbol : ký hiệu

segment_type : loại *segment*

Ký hiệu là tên của một *segment* tái định vị. Trong việc sử dụng các *segment*, ASM51 là trình dịch hợp ngữ phức tạp hơn các trình dịch hợp ngữ qui ước ( tổng quát chỉ hỗ trợ hai loại *segment* là " code " và " data " ). ASM51 định nghĩa thêm các loại *segment* sao cho phù hợp với các không gian nhớ khác nhau trong hệ thống sử dụng 8051. Dưới đây là các loại *segment* ( các không gian bộ nhớ ) được định nghĩa cho 8051 :

- CODE ( *segment* chương trình hay *segment* mã )
- XDATA ( không gian dữ liệu ngoài )
- DATA ( không gian dữ liệu nội được truy xuất bởi kiểu định địa chỉ trực tiếp, 00H – 7FH )
- IDATA ( toàn thể không gian dữ liệu nội được truy xuất bởi kiểu định địa chỉ gián tiếp, 00H – 7FH, 00H – FFH đối với 8052 ).
- BIT ( không gian bit; không gian này chiếm các địa chỉ byte từ 20H đến 2FH trong không gian dữ liệu nội ).

Thí dụ phát biểu sau :

EPROM SEGMENT CODE

khai báo ký hiệu EPROM là tên của SEGMENT loại CODE. Lưu ý là phát biểu này chỉ đơn giản khai báo EPROM là gì. Trên thực tế để bắt đầu sử dụng *segment* này, ta phải sử dụng chỉ dẫn RSEG ( đề cập sau ).

#### 7.5.2.2 EQU ( equate )

Dạng của chỉ dẫn EQU như sau :

Symbol EQU expression

symbol : ký hiệu

expression : biểu thức

Chỉ dẫn EQU gán giá trị số cho tên của ký hiệu được định nghĩa. Ký hiệu phải có tên hợp lệ và biểu thức phải tuân theo các qui luật đã được mô tả trước đây.

Dưới đây là các thí dụ cho chỉ dẫn EQU :

```
N27        EQU   27            ; N27 có giá trị 27
HERE       EQU   $             ; HERE có giá trị của bộ đếm vị trí
CR         EQU   0DH           ; CR có giá trị 0DH ( xuống dòng )
MESSAGE    DB    ' This is a message '
LENGTH     EQU   $ - MESSAGE
                               ; 'LENGTH' có giá trị là chiều dài ;
                               của MESSAGE
```

### 7.5.2.3 Các chỉ dẫn định nghĩa ký hiệu khác

Chỉ dẫn SET tương tự như chỉ dẫn EQU ngoại trừ ký hiệu sau đó có thể được định nghĩa lại bằng cách sử dụng chỉ dẫn SET khác.

Các chỉ dẫn DATA, IDATA, XDATA, BIT và CODE gán các địa chỉ của loại *segment* tương ứng cho một ký hiệu. Các chỉ dẫn này không quan trọng. Kết quả tương tự có thể nhận được bằng cách sử dụng chỉ dẫn EQU, tuy nhiên nếu được sử dụng chúng sẽ cho ta phương tiện kiểm tra loại ( type ) bởi ASM51.

Ta hãy khảo sát 2 chỉ dẫn và 4 lệnh sau :

```
FLAG1      EQU    05H
FLAG2      BIT    05H
           SETB   FLAG1
           SETB   FLAG2
           MOV    FLAG1, #0
           MOV    FLAG2, #0
```

Việc sử dụng FLAG2 ở lệnh sau cùng trong chuỗi lệnh trên sẽ tạo ra một thông báo sai từ ASM51 " data segment address expected ". Do FLAG2 được định nghĩa như là một địa chỉ bit ( bằng cách sử dụng chỉ dẫn BIT ), FLAG2 có thể được dùng trong các lệnh liên quan đến bit nhưng không thể được sử dụng trong các lệnh liên quan đến byte nên tạo ra lỗi. Mặc dù FLAG1 có cùng giá trị là 05H, FLAG1 được định nghĩa bởi EQU và không có một không gian địa chỉ nào được kết hợp.

Đây không phải là ưu điểm mà là khuyết điểm của EQU. Bằng cách định nghĩa rõ ràng các ký hiệu địa chỉ dùng cho không gian nhớ cụ thể ( sử dụng các chỉ dẫn BIT, DATA, XDATA v.v... ), người lập trình tận dụng được ưu điểm của việc kiểm tra loại của ASM51 và tránh được các phiền toái do sử dụng sai các ký hiệu.

### 7.5.3 Khởi động và dành trước vùng nhớ

Các chỉ dẫn khởi động và dành trước vùng nhớ sẽ khởi động và dành trước không gian nhớ tính bằng từ, byte hoặc bit. Không gian được dành trước bắt đầu ở vị trí được chỉ ra bởi giá trị hiện hành của bộ đếm vị trí *trong segment* tích cực hiện hành. Các chỉ dẫn này có thể có một nhãn đứng trước. Chúng được mô tả như sau :

#### 7.5.3.1 DS ( define storage )

Dạng của chỉ dẫn định nghĩa vùng nhớ DS như sau :

```
[label:]        DS      expression
```

label : nhãn

expression : biểu thức

Chỉ dẫn DS dành trước vùng nhớ tính bằng byte. Chỉ dẫn này được dùng cho bất kỳ loại *segment* nào trừ BIT. Biểu thức phải là biểu thức thời gian dịch hợp lệ không có tham chiếu thuận và không tái định vị hoặc các tham chiếu ngoài. Khi gặp phát biểu sử dụng DS trong chương trình, bộ đếm vị trí của *segment* hiện hành được tăng bởi giá trị của biểu thức. Tổng của nội dung bộ đếm vị trí với giá trị của biểu thức không được vượt quá giới hạn của không gian địa chỉ hiện hành.

Các phát biểu sau tạo ra một vùng đệm 40-byte trong *segment* dữ liệu nội :

```
                DSEG    AT      30H     ; đặt trong segment dữ liệu
                                        ; (tuyệt đối, nội)
LENGTH:         EQU     40
BUFFER:         DS      LENGTH          ; dành trước 40 byte
```

Nhãn BUFFER biểu diễn địa chỉ của vị trí đầu tiên của vùng nhớ được dành trước. Với thí dụ này, vùng đệm bắt đầu ở địa chỉ 30H do " AT 30H " được xác định bởi DSEG ( xem mục 7.5.5.2 : lựa chọn các *segment* tuyệt đối ). Vùng đệm này có thể được xóa bằng cách sử dụng chuỗi lệnh sau :

```
        MOV     R7, #LENGTH
        MOV     R0, #BUFFER
LOOP:   MOV     @R0, #0
        DJNZ    R7, LOOP
```

Để tạo ra vùng đệm 1000 byte trong bộ nhớ ngoài ở địa chỉ bắt đầu là 4000H, ta có thể sử dụng các chỉ dẫn sau :

```
XSTART:         EQU     4000H
XLENGTH:        EQU     1000
                XSEG    AT      XSTART
XBUFFER:        DS      XLENGTH
```

Bộ đệm này có thể được xóa bằng chuỗi lệnh sau :

```
                MOV     DPTR, #XBUFFER
LOOP:           CLR     A
                MOVX    @DPTR, A
                INC     DPTR
                MOV     A, DPL
                CJNE    A, #LOW ( XBUFFER + XLENGTH + 1 ), LOOP
                MOV     A, DPH
                CJNE    A, #HIGH ( XBUFFER + XLENGTH + 1 ), LOOP
```

Đây là thí dụ minh họa việc sử dụng các toán tử của ASM51 và các biểu thức thời gian dịch. Vì không có lệnh nào tồn tại để so sánh nội dung con trỏ dữ liệu với một giá trị tức thời, thao tác này phải được tạo ra từ các lệnh hợp lệ. Có hai lệnh so sánh được cần đến, một cho byte thấp và một cho byte cao của DPTR.

Thêm nữa, lệnh " so sánh và nhảy nếu không bằng " chỉ làm việc với thanh chứa hoặc thanh ghi, do vậy các byte của con trỏ dữ liệu phải được di chuyển đến thanh chứa trước khi sử dụng lệnh CJNE. Vòng lặp kết thúc chỉ khi con trỏ dữ liệu đạt được giá trị bằng với XBUFFER + XLENGTH + 1 ( 1 được cần đến là do con trỏ dữ liệu được tăng sau khi có lệnh MOVX sau cùng ).

### 7.5.3.2 DBIT

Dạng của chỉ dẫn DBIT như sau :

[label:]    DBIT    expression

label : nhãn

expression : biểu thức

Chỉ dẫn DBIT dành trước vùng nhớ tính bằng bit. Chỉ dẫn này chỉ có thể sử dụng với *segment* BIT. Biểu thức phải là biểu thức thời gian dịch hợp lệ và không có tham chiếu thuận. Khi gặp phát biểu sử dụng DBIT trong chương trình, bộ đếm vị trí của *segment* ( BIT ) hiện hành được tăng bởi giá trị của biểu thức. Lưu ý là trong *segment* BIT, đơn vị

của bộ đếm vị trí là bit thay vì là byte. Các chỉ dẫn sau tạo ra 3 cờ trong một *segment* bit tuyệt đối :

```
          BSEG            ; segment bit tuyệt đối
KBFLAG:   DBIT   1        ; trạng thái của bàn phím
PRFLAG:   DBIT   1        ; trạng thái của máy in
DKFLAG:   DBIT   1        ; trạng thái của đĩa
```

Vì không có địa chỉ được xác định cho BSEG trong thí dụ trên, địa chỉ của các cờ được định nghĩa bằng DBIT có thể được xác định ( nếu muốn ) bằng cách khảo sát bảng các ký hiệu trong các tập tin .LST hoặc .M51. ( xem hình 7.1 và hình 7.5 ). Nếu các định nghĩa ở trên sử dụng BSEG lần đầu tiên, cờ KBFLAG được đặt ở địa chỉ bit là 00H ( bit 0 của byte có địa chỉ 20H ). Nếu có các bit khác được định nghĩa trước bằng cách sử dụng BSEG, các định nghĩa ở trên tạo ra các cờ có địa chỉ tiếp theo sau địa chỉ bit sau cùng của các định nghĩa đã có trước. ( xem mục 7.5.5.2 ).

### 7.5.3.3 DB ( define byte )

Dạng của chỉ dẫn DB ( định nghĩa byte ) như sau :

[label:]  DB  expression [, expression] [ . . . . ]

label : nhãn

expression : biểu thức

Chỉ dẫn DB khởi động vùng nhớ mã cùng với các giá trị của byte. Do trên thực tế chỉ dẫn này thường được dùng để đặt các hằng số vào bộ nhớ chương trình, một *segment* CODE phải được tích cực. Danh sách các biểu thức là một chuỗi của một hay nhiều giá trị byte ( mỗi một giá trị byte có thể là một biểu thức ) cách nhau bởi dấu phẩy.

Chỉ dẫn DB cho phép chuỗi ký tự ( đặt trong hai dấu nháy đơn ) dài hơn 2 ký tự miễn sao chúng không phải là một phần của biểu thức. Mỗi một ký tự trong chuỗi được biến đổi thành mã ASCII tương ứng. Nếu có một nhãn được dùng, nhãn được gán địa chỉ của byte đầu tiên. Thí dụ các phát biểu sau :

```
           CSEG   AT     0100H
SQUARES:   DB     0, 1, 4, 9, 16, 25     ; bình phương
MESSAGE:   DB     'Login:', 0            ; chuỗi ký tự kết
                                         ; thúc bởi 0
```

khi được hợp dịch sẽ tạo ra kết quả gán cho bộ nhớ chương trình ngoài ở dạng các số hex sau :

| **Địa chỉ** | **Nội dung** |
|---|---|
| 100 | 00 |
| 101 | 01 |
| 102 | 04 |
| 103 | 09 |
| 104 | 10 |
| 105 | 19 |
| 106 | 4C |
| 107 | 6F |
| 108 | 67 |
| 109 | 69 |
| 10A | 6E |
| 10B | 3A |
| 10C | 00 |

#### 7.5.3.4 DW ( define word )

Dạng của chỉ dẫn DW ( định nghĩa từ ) như sau :

[label:] DW expression [, expression] [ . . . . ]

label : nhãn

expression : biểu thức

Chỉ dẫn DW tương tự như chỉ dẫn DB ngoại trừ hai vị trí nhớ ( 16 bit ) được gán cho mỗi một thành phần dữ liệu. Thí dụ các phát biểu :

```
CSEG    AT      200H
DW      $, 'A', 1234H, 2. 'BC'
```

sẽ cho ra kết quả như sau :

| **Địa chỉ** | **Nội dung** |
|---|---|
| 200 | 02 |
| 201 | 00 |
| 202 | 00 |

| | |
|---|---|
| 203 | 41 |
| 204 | 12 |
| 205 | 34 |
| 206 | 00 |
| 207 | 02 |
| 208 | 42 |
| 209 | 43 |

### 7.5.4 Liên kết chương trình

Các chỉ dẫn liên kết chương trình cho phép các mô-đun ( các tập tin ) được hợp dịch riêng rẽ truyền thông với nhau bằng cách cho phép các tham chiếu liên mô-đun ( intermodule ) và đặt tên cho các mô-đun. Trong phần sau đây, một mô-đun được xem như là một tập tin ( trong thực tế một mô-đun có thể bao gồm nhiều hơn một tập tin ).

#### 7.5.4.1 PUBLIC

Dạng của chỉ dẫn PUBLIC như sau :

PUBLIC symbol [, symbol] [ . . . . ]

symbol : ký hiệu

Chỉ dẫn PUBLIC cho phép một danh sách các ký hiệu đã được xác định được biết và được sử dụng ở bên ngoài của mô-đun được hợp dịch hiện hành ( nghĩa là được biết và được sử dụng bởi các mô-đun khác với mô-đun chứa danh sách các ký hiệu nêu trên ). Như vậy một ký hiệu được khai báo bởi PUBLIC phải được định nghĩa trong mô-đun hiện hành và được tham chiếu bởi một mô-đun khác. Thí dụ :

PUBLIC INCHAR, OUTCHR, INLINE, OUTSTR

#### 7.5.4.2 EXTRN

Dạng của chỉ dẫn EXTRN như sau :

EXTRN segment_type ( symbol [, symbol] [ . . . . ] . . . . )

segment_type : loại *segment*

symbol : ký hiệu

Chỉ dẫn EXTRN liệt kê các ký hiệu được tham chiếu trong mô-đun hiện hành, các ký hiệu này được định nghĩa trong các mô-đun khác. Danh sách của các ký hiệu bên ngoài này phải có loại *segment* kết hợp với mỗi một ký hiệu trong danh sách.

Loại *segment* là CODE, XDATA, DATA, IDATA. BIT và NUMBER. NUMBER là một ký hiệu được gán bởi EQU.

Loại *segment* chỉ ra cách thức một ký hiệu được sử dụng. Thông tin này quan trọng ở thời gian liên kết để đảm bảo các ký hiệu được sử dụng đúng trong các mô-đun khác.

Các chỉ dẫn PUBLIC và EXTRN hoạt động cùng nhau. Ta hãy khảo sát hai tập tin dưới đây, MAIN.SRC và MESSAGES.SRC. Các chương trình con HELLO và GOOD_BYTE được định nghĩa trong mô-đun MESSAGES nhưng được gọi bởi các mô-đun khác bằng cách sử dụng chỉ dẫn PUBLIC.

Các chương trình con được gọi trong mô-đun MAIN mặc dù chúng không được định nghĩa ở đây. Chỉ dẫn EXTRN khai báo các ký hiệu này được định nghĩa trong mô-đun khác.

```
MAIN.SRC
                EXTRN CODE ( HELLO, GOOD_BYTE )
                . . . . . .
                CALL   HELLO
                . . . . . .
                CALL   GOOD_BTYE
                . . . . . .
                END
MESSAGES.SRC
                PUBLIC        HELLO, GOOD_BYTE
                . . . . . .
     HELLO:     ( bắt đầu chương trình con )
                . . . . . .
                RET
     GOOD_BYTE: ( bắt đầu chương trình con )
                . . . . . .
                RET
```

Cả hai mô-đun MAIN.SRC và MESSAGES.SRC đều không phải là các chương trình hoàn chỉnh; chúng được hợp dịch riêng rẽ và được liên kết với nhau để tạo ra một chương trình thực thi được. Trong quá trình

liên kết, các tham chiếu ngoài được tính toán các địa chỉ đúng để chèn vào đích gọi của các lệnh CALL.

#### 7.5.4.3 NAME

Dạng của chỉ dẫn NAME như sau :

NAME module_name

Module_name : tên của mô-đun

Tất cả các qui luật áp dụng cho tên của ký hiệu cũng áp dụng cho tên của mô-đun. Nếu tên của mô-đun không hiện diện, mô-đun sẽ lấy tên của tập tin ( nhưng không có các định danh ổ đĩa, thư mục và không có phần mở rộng ). Nếu vắng mặt chỉ dẫn NAME, một chương trình sẽ chứa một mô-đun cho mỗi một tập tin.

Khái niệm " mô-đun " có hơi nặng nề đối với các vấn đề lập trình tương đối nhỏ. Ngay cả đối với các chương trình có kích thước trung bình ( thí dụ bao gồm một vài tập tin cùng với các *segment* tái định vị ) ta cũng không cần sử dụng chỉ dẫn NAME và cũng không cần chú ý đến khái niệm " mô-đun ".

Tuy nhiên với các chương trình rất lớn ( vài ngàn dòng lệnh hoặc hơn ), ta cần phải chia vấn đề ra làm nhiều mô-đun trong đó thí dụ mỗi một mô-đun có thể bao gồm vài tập tin chứa các chương trình có cùng mục đích.

### 7.5.5 Các chỉ dẫn lựa chọn *segment*

Khi trình dịch hợp ngữ gặp một chỉ dẫn lựa chọn *segment*, trình dịch hợp ngữ sẽ hướng về mã hoặc dữ liệu theo sau trong *segment* được lựa chọn cho đến khi một *segment* khác được lựa chọn bởi chỉ dẫn lựa chọn *segment*. Chỉ dẫn này có thể lựa chọn một *segment* tái định vị đã được định nghĩa trước đó hoặc tạo ra và lựa chọn các *segment* tuyệt đối.

#### 7.5.5.1 RSEG ( relocatable segment )

Dạng của chỉ dẫn RSEG như sau :

RSEG segment_name

Segment_name : tên *segment*

Trong đó tên *segment* là tên của *segment* tái định vị đã được định nghĩa trước bằng chỉ dẫn SEGMENT. RSEG là một chỉ dẫn lựa chọn *segment* hướng về chuỗi mã hoặc dữ liệu tiếp theo trong *segment* đã được đặt tên cho đến khi trình dịch hợp ngữ gặp một chỉ dẫn lựa chọn *segment* khác.

#### 7.5.5.2 Việc lựa chọn các *segment* tuyệt đối

RSEG chọn một *segment* tái định vị. Các *segment* tuyệt đối được lựa chọn bằng cách sử dụng các chỉ dẫn sau :

CSEG [ AT address ]

DSEG [ AT address ]

ISEG [ AT address ]

BSEG [ AT address ]

XSEG [ AT address ]

address : địa chỉ

Các chỉ dẫn này lựa chọn một *segment* tuyệt đối trong không gian địa chỉ của chương trình, dữ liệu nội, dữ liệu nội gián tiếp, bit hoặc dữ liệu bên ngoài. Nếu một địa chỉ tuyệt đối được xác định bằng cách dùng " AT address ", trình dịch hợp ngữ kết thúc *segment* địa chỉ tuyệt đối sau cùng ( nếu có ) thuộc loại *segment* được xác định và bắt đầu một *segment* tuyệt đối mới tại địa chỉ đã chỉ ra. Nếu địa chỉ tuyệt đối không được xác định, *segment* tuyệt đối sau cùng thuộc loại được xác định sẽ được tiếp tục. Nếu không có *segment* tuyệt đối nào thuộc loại này được lựa chọn trước đó và địa chỉ tuyệt đối được bỏ qua, một *segment* mới được tạo ra và bắt đầu ở địa chỉ là 0. Các tham chiếu thuận không được phép và các địa chỉ bắt đầu phải là địa chỉ tuyệt đối.

Mỗi một *segment* có một bộ đếm vị trí riêng và bộ đếm này thường được khởi động bằng 0. *Segment* mặc định là *segment* chương trình tuyệt đối; do vậy trạng thái bắt đầu của trình dịch hợp ngữ sẽ ở vị trí 0000H trong *segment* chương trình tuyệt đối. Khi một *segment* khác được chọn lần đầu tiên, bộ đếm vị trí của *segment* trước duy trì giá trị tích cực sau cùng. Khi *segment* trước được chọn lại lần nữa, bộ đếm vị trí lấy lại giá trị tích cực sau cùng nêu trên.

| LOC | OBJ | LINE | SOURCE | | | |
|---|---|---|---|---|---|---|
| | | 1 | ONCHIP | SEGMENT | DATA | |
| | | 2 | EPROM | SEGMENT | CODE | |
| | | 3 | | | | |
| .... | | 4 | | BSEG | AT | 70H |
| 0070 | | 5 | FLAG1: | DBIT | 1 | |
| 0071 | | 6 | FLAG2: | DBIT | 1 | |

Hình 7.3 : Định nghĩa và khởi động các *segment* tái định vị và tuyệt đối

| | | | | | |
|---|---|---|---|---|---|
| | | 7 | | | |
| .... | | 8 | | RSEG | ONCHIP |
| 0000 | | 9 | TOTAL: | DS | 1 |
| 0001 | | 10 | COUNT: | DS | 1 |
| 0002 | | 11 | SUM16: | DS | 2 |
| | | 12 | | | |
| .... | | 13 | | RSEG | EPROM |
| 0000 | 750000 F | 14 | BEGIN: | MOV | TOTAL, #0 |
| | | 15 | ; | ( continue program ) | |
| | | 16 | END | | |

Hình 7.3 : ( tiếp theo )

LOC : vị trí

OBJ : mã đối tượng

LINE : dòng

SOURCE : mã nguồn

Chỉ dẫn ORG được dùng để thay đổi giá trị của bộ đếm vị trí trong *segment* được chọn hiện hành. Hình 7.3 trình bày một thí dụ về việc định nghĩa và khởi động các *segment* tái định vị và tuyệt đối.

Hai dòng đầu tiên ở hình 7.3 khai báo các ký hiệu ONCHIP và EPROM như là hai *segment* loại DATA ( RAM dữ liệu nội ) và CODE. Dòng 4 bắt đầu một *segment* bit tuyệt đối, địa chỉ bit bắt đầu là 70H ( bit 0 của byte địa chỉ 2EH ). Kế đến các cờ FLAG1 và FLAG2 được tạo ra dưới dạng các nhãn tương ứng với các vị trí có địa chỉ bit là 70H và 71H. RSEG ở dòng 8 bắt đầu *segment* ONCHIP tái định vị của RAM dữ liệu nội. TOTAL và COUNT là các nhãn có các địa chỉ byte tương ứng như trong hình. SUM16 là nhãn tương ứng với vị trí từ ( 2 byte ). Sự xuất hiện lần nữa của RSEG ở dòng 13 bắt đầu *segment* EPROM tái định vị của bộ nhớ chương trình. Nhãn BEGIN có địa chỉ của lệnh đầu tiên trong thể hiện ( instance ) này của EPROM. Lưu ý là ta không thể xác định được địa chỉ của các nhãn TOTAL, COUNT, SUM16 và BEGUN từ hình 7.3 do các nhãn này xuất hiện trong các *segment* tái định vị, tập tin đối tượng ( object file ) phải được xử lý bởi trình liên kết và định vị ( xem mục 7.7 ) với các địa chỉ bắt đầu được chỉ rõ cho các *segment* ONCHIP và EPROM. Tập tin liệt kê .M51 được tạo ra bởi trình liên kết và định vị cho ta các địa chỉ tuyệt đối của các nhãn vừa nêu trên. FLAG1 và FLAG2 luôn luôn có địa chỉ bit là 70H và 71H do chúng được định nghĩa trong *segment* BIT tuyệt đối.

## 7.6 CÁC ĐIỀU KHIỂN CỦA TRÌNH DỊCH HỢP NGỮ

Các điều khiển của trình dịch hợp ngữ thiết lập khuôn dạng cho các tập tin đối tượng và tập tin liệt kê bằng cách điều hòa các hoạt động của ASM51. Thông thường các điều khiển của trình dịch hợp ngữ ảnh hưởng đến cách trình bày tập tin liệt kê, không ảnh hưởng đến bản thân chương trình. Các điều khiển có thể được đưa vào dòng lệnh khi ta cho chương trình được hợp dịch hoặc chúng có thể được đặt trong tập tin nguồn. Các điều khiển của trình dịch hợp ngữ khi xuất hiện trong tập tin nguồn phải được đứng trước bởi dấu $ và phải được bắt đầu ở cột 1.

Có hai loại điều khiển của trình dịch hợp ngữ : loại cơ bản P và loại tổng quát G. Các điều khiển cơ bản được đặt trong dòng lệnh khi ta cho chương trình được hợp dịch hoặc đặt ở điểm bắt đầu của chương trình nguồn. Chỉ có các điều khiển cơ bản mới được đặt trước một điều khiển cơ bản. Các điều khiển tổng quát có thể đặt ở bất kỳ nơi nào trong chương trình nguồn. Hình 7.4 trình bày các điều khiển thông dụng của trình dịch hợp ngữ ASM51.

| Tên | P/G | Mặc định | Ký hiệu | Ý nghĩa |
|---|---|---|---|---|
| DATE ( date ) | P | DATE( ) | DA | Đặt chuỗi trong header ( 9 ký tự ) |
| DEBUG | P | NODEBUG | DB | Xuất thông tin ký hiệu debug cho tập tin đối tượng |
| NODEBUG | P | NODEBUG | NODB | Thông tin ký hiệu debug không đặt trong tập tin đối tượng |
| ERRORPRINT | P | NOERRORPRINT | EP | Chỉ định một tập tin nhận các thông báo lỗi cùng với tập tin liệt kê. |
| NOERRORPRINT | P | NOERRORPRINT | NOEP | Chỉ định rằng các thông báo lỗi sẽ chỉ được in trong tập tin liệt kê. |
| GEN | G | GENONLY | GE | Tạo ra một danh sách đầy đủ các quá trình mở rộng *macro* bao gồm các lời gọi *macro* và đưa vào tập tin liệt kê |
| GENONLY | G | GENONLY | GO | Chỉ liệt kê mã nguồn được mở rộng đầy đủ y như các dòng được tạo ra bởi lời gọi *macro* đã ở trong tập tin nguồn. |

**Hình 7.4 : Các điều khiển thông dụng được hỗ trợ bởi trình dịch hợp ngữ**

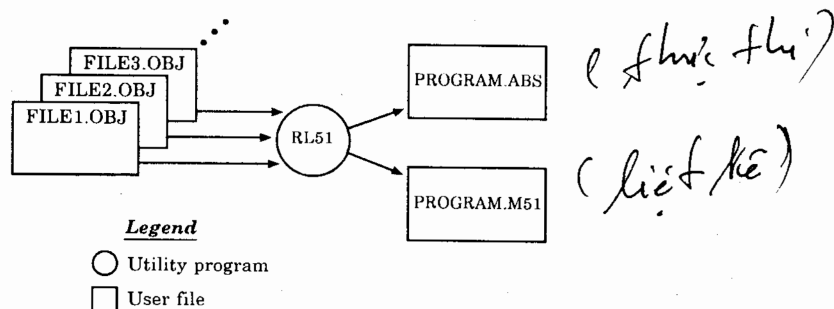
| | | | | |
|---|---|---|---|---|
| NOGEN | G | GENONLY | NOGE | Chỉ liệt kê tập tin nguồn ban đầu trong tập tin liệt kê |
| INCLUDE | G | không áp dụng | IC | Chỉ định rằng một tập tin được bao gồm trong một chương trình |
| LIST | G | LIST | LI | In các dòng liên tiếp nhau của mã nguồn trong tập tin liệt kê |
| NOLIST | G | LIST | NOLI | Không in các dòng liên tiếp nhau của mã nguồn trong tập tin liệt kê |
| MACRO | P | MACRO | MR | Đánh giá và mở rộng tất cả các lời gọi *macro*. Cấp phát phần trăm bộ nhớ còn trống cho việc xử lý *macro* |
| NOMACRO | P | MACRO | NOMR | Không đánh giá các lời gọi *macro* |
| MO051 | P | MO051 | MO | Nhận biết các SFR tiền định nghĩa của 8051 |
| NOMO051 | P | MO051 | NOMO | Không nhận biết các SFR tiền định nghĩa của 8051 |
| OBJECT [(file)] | P | OBJECT | OJ | Chỉ rõ tập tin nhận mã đối tượng |
| NOOBJECT | P | OBJECT | NOOJ | Không có tập tin đối tượng tạo ra |
| PAGING | P | PAGING | PI | Tập tin liệt kê tách làm nhiều trang, mỗi trang có một header |
| NOPAGING | P | PAGING | NOPI | Tập tin liệt kê không tách ra nhiều trang |
| PAGELENGTH(n) | P | PAGELENGTH(n) | PL | Thiết lập số dòng tối đa cho một trang ( 10 – 65535 ) |
| PAGEWIDTH(n) | P | PAGEWIDTH(n) | PW | Thiết lập số ký tự tối đa cho một dòng ( 72 – 132 ) |
| PRINT [(file)] | P | PRINT | PR | Chỉ rõ tên tập tin nhận liệt kê |
| NOPRINT | P | PRINT | NOPR | Không có tập tin liệt kê tạo ra |
| SYMBOLS | P | SYMBOLS | SB | Tạo ra bảng tất cả các ký hiệu |
| NOSYMBOLS | P | SYMBOLS | NOSB | Không có bảng ký hiệu tạo ra |
| TITLE(string) | G | TITLE() | TT | Đặt một chuỗi vào tất cả các header của trang (60 ký tự tối đa) |

**Hình 7.4 : Các điều khiển thông dụng được hỗ trợ bởi trình dịch hợp ngữ**

## 7.7 HOẠT ĐỘNG LIÊN KẾT

Khi phát triển các chương trình ứng dụng lớn, ta thường chia các công việc thành các chương trình con hoặc các mô-đun chứa các phần mã ( section of code ) ( thường là các chương trình con ), chúng được viết độc lập. Thuật ngữ lập trình theo mô-đun ( modular programming ) dùng để chỉ chế độ lập trình này. Một cách tổng quát các mô-đun thuộc loại tái định vị nghĩa là chúng không được dự kiến có các địa chỉ cụ thể trong không gian chương trình hoặc dữ liệu. Một trình liên kết và định vị được cần đến để kết hợp các mô-đun này thành một mô-đun đối tượng tuyệt đối và thực thi được.

RL51 của Intel là một trình liên kết và định vị điển hình. Trình này xử lý một chuỗi các mô-đun đối tượng tái định vị ở ngõ vào và tạo ra một chương trình ngôn ngữ máy thực thi được ( chẳng hạn PRO-GRAM ) và một tập tin liệt kê chứa một bản đồ bộ nhớ và bảng ký hiệu ( PROGRAM.M51 ). Điều này được minh họa trong hình 7.5.




Hình 7.5 : Hoạt động liên kết

Khi các mô-đun tái định vị được kết hợp, tất cả các giá trị của các ký hiệu ngoài được tính toán và chèn giá trị tính được vào tập tin kết quả. Trình liên kết được gọi từ dấu nhắc hệ thống như sau :

RL51 input_list [ TO output_file ] [ location_controls ]

input_list : danh sách ngõ vào

output_file : tập tin kết quả

location_controls : các điều khiển vị trí

Danh sách ngõ vào là danh sách các mô-đun ( hay tập tin ) đối tượng tái định vị được phân cách bởi dấu phẩy. Tập tin kết quả là tên của mô-đun đối tượng tuyệt đối được tạo ra. Nếu tên của mô-đun này không được cung cấp, tên sẽ là tên của tập tin ngõ vào đầu tiên không có phần mở rộng. Các điều khiển vị trí thiết lập các địa chỉ bắt đầu cho các *segment* đã được đặt tên.

Thí dụ, giả sử ta có 3 mô-đun hoặc tập tin là MAIN.OBJ, MESSAGES.OBJ và SUBROUTINES.OBJ, chúng được kết hợp để tạo ra chương trình thực thi được EXAMPLE và mỗi một mô-đun vừa nêu trên chứa 2 *segment* tái định vị, một gọi là EPROM thuộc loại CODE và một gọi là ONCHIP thuộc loại DATA. Ta cũng giả sử thêm rằng *segment* chương trình được thực thi ở địa chỉ 4000H và *segment* dữ liệu được đặt ở địa chỉ bắt đầu là 30H trong RAM nội. Dòng lệnh sau đây có thể được sử dụng :

```
RL51   MAIN.OBJ, MESSAGES.OBJ, SUBROUTINES.OBJ  TO  EXAMPLE &
CODE   ( EPROM ( 4000H ) )  DATA ( ONCHIP ( 30H ) )
```

Lưu ý là ký tự "&" được dùng để chỉ ra rằng dòng lệnh còn được tiếp tục.

Nếu chương trình bắt đầu ở nhãn START và đây là lệnh đầu tiên trong mô-đun MAIN, việc thực thi sẽ bắt đầu ở địa chỉ 4000H. Nếu mô-đun MAIN không phải là thành phần được liệt kê trước tiên trong lệnh ( nghĩa là không được liên kết đầu tiên ) hoặc nếu nhãn START không phải là nơi bắt đầu của MAIN, điểm nhập của chương trình có thể được xác định bằng cách khảo sát bảng ký hiệu trong tập tin liệt kê EXAMPLE.LST được tạo ra bởi RL51.

Nếu mặc định, EXAMPLE.LST chỉ chứa bản đồ liên kết. Nếu bảng ký hiệu được cần đến, mỗi một chương trình nguồn phải sử dụng điều khiển $DEBUG ( xem mục 7.5 ).

## 7.8 THÍ DỤ

Nhiều khái niệm vừa được giới thiệu được đưa vào thí dụ này, đây là một chương trình đơn giản viết cho 8051. Mã nguồn được chia thành 2 tập tin và sử dụng các ký hiệu được khai báo bằng các chỉ dẫn PUBLIC, EXTRN để cho phép truyền thông liên tập tin ( inter-file ).

Mỗi một tập tin là một mô-đun – một mô-đun mang tên MAIN và mô-đun kia mang tên SUBROUTINES. Chương trình sử dụng *segment* mã tái định vị có tên là EPROM và *segment* dữ liệu nội tái định vị có tên là ONCHIP. Làm việc với nhiều tập tin, mô-đun và *segment* là điều cần thiết để có các dự án lập trình lớn.

Việc khảo sát cẩn thận thí dụ trong mục này sẽ giúp ta tăng cường hiểu biết về các khái niệm cốt lõi này cũng như giúp ta chuẩn bị tham gia vào các thiết kế thực tế dựa trên 8051. Thí dụ ở mục này là một chương trình xuất nhập đơn giản sử dụng *port* nối tiếp của 8051 cùng với một bàn phím và một màn hình hiển thị CRT của một thiết bị đầu cuối hiển thị *video* VDT.

```
MSC-51 MACRO ASSEMBLER   *** EXAMPLES (MAIN MODULE) ***      03/17/91

DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN ECHO.OBJ
ASSEMBLER INVOKED BY : C:\ASM\ASM51.EXE  ECHO.SCR  EP

LOC    OBJ         LINE   SOURCE
                     1    $DEBUG
                     2    $TITLE ( *** EXAMPLES (MAIN MODULE)*** )
                     3    $PAGEWIDTH(98)
                     4    $NOPAGING
                     5
                     6           NAME    MAIN                ;MODULE NAME IS "MAIN"
                     7           EXTRN CODE(INIT,OUTSTR)     ;DECLARE       EXTERNAL
SYMBOLS
                     8           EXTRN CODE(INLINE,OUTLINE)
                     9
0000                10    CR     EQU     0DH                 ;CARRIAGE RETURN CODE
                    11    EPROM SEGMENT CODE                 ;DEFINE  SYMBOL  "EPROM
                    12
----                13           RSEG    EPROM               ;BEGIN CODE SEGMENT
0000 120000   F     14    MAIN:  CALL    INIT                ;INITIALIZE SERIAL PORT
0003 900000   F     15    LOOP:  MOV     DPTR,#PROMPT        ;SENO PROMPT
0006 120000   F     16           CALL    OUTSTR
0009 120000   F     17           CALL    INLINE              ;GET A COMMAND LINE
000C 120000   F     18           CALL    OUTLINE             ;AND ECHO IT BACK
000F 80F2           19           JMP     LOOP                ;REPEAT
                    20
0011 0D             21    PROMPT: DB     CR,'Enter a command :',0
0012 456E7465
0016 72206120
001A 636F6060
001E 616E643A
0022 20
0023 00
                    22           END
SYMBOL TABLE LISTING
------ -------------
NAME      TYPE     VALUE    ATTRIBUTES
CR . . . .  NUMB     0000H  A
EPROM. . .  C SEG    0024H    REL=UNIT
INIT . . .  C ADDR   ----   EXT
INLINE . .  C ADDR   ----   EXT
```

Hình 7.7 : Thí dụ : liên kết các segment và mô-đun tái định vị (a) ECHO.LST (b) IO.LST (c) EXAMPLE.M5

```
LOOP . . .      C ADDR    0003H  R      SEG=EPROM
MAIN . . .      C ADDR    0000H  R      SEG=EPROM
OUTLINE. .      C ADDR    - - - -     EXT
OUTSTR . .      C ADDR    - - - -     EXT
PROMPT . .      C ADDR    0011H  R      SEG=EPROM

REGISTER BANK(S) USED: 0
ASSEMBLY COMPLETE, NO ERRORS FOUND


MSC-51 MACROASSEMBLER *** ANNOTATED EXAMPLES (SUBROUTINES MODULE)
***    03/17/91

DOS 3.31 (038-N) MCS-51 MACRO ASSEMBLER, V2.2
OBJECT MODULE PLACED IN IO.OBJ
ASSEMBLER INVOKED BY : C:\ASM51\ASM51.EXE  IO.SCR  EP

LOC    OBJ        LINE   SOURCE
                    1    $DEBUG
                    2    $TITLE ( *** EXAMPLES (SUBROUTINES MODULE)*** )
                    3    $PAGEWIDTH(98)
                    4    $NOPAGING
                    5
                    6    NAME   SUBROUTINES              ;MODULE NAME
                    7    PUBLIC INIT,OUTCHR,INCHAR       ;DECLARE PUBLIC SYMBOLS
                    8    PUBLIC INLINE,OUTLINE,OUTSTR
                    9
                   10    ;************************************************************
                   11;   DEFINE SYMBOLS
                   12    ;************************************************************
0000               13    CR       EQU       0DH        ;CARRIAGE RETURN
0028               14    LENGTH   EQU       40         ;40-CHARACTER BUFFER
                   15    EPROM    SEGMENT   CODE       ;"EPROM" IS A CODE SEGMENT
                   16    ONCHIP   SEGMENT   DATA       ;"ONCHIP" IS A DATA SEGMENT
                   17
- - - -            18             RSEG      EPROM      ;BEGIN RELOCATABLE CODE
                                                       ;SEG
                   19
                   20    ;************************************************************
                   21    ; INITIALIZE THE SERIAL PORT
                   22    ;************************************************************
0000 759852        23    INIT:  MOV   SCON,#52H        ;8-BIT UART MODE
0003 758920        24           MOV   TMOD,#20H        ;TIMER1 SUPPLIES BAUD RATE
                                                       ;CLOCK
```

Hình 7.7 : ( tiếp theo )

```
0006 758DF3              MOV     TH1,#-13        ;2400 BAUD
0009 028E                SETB    TR1             ;START TIMER
000B 22                  RET

                ;*************************************************************
                ; OUTPUT CHARACTER IN ACC (NOTE: VDT MUST CONVERT CR
                ; INTO CR/LF)
                ;*************************************************************
000C 3099FD     OUTCHR:  JNB     TI,$            ;WAIT FOR TRANSMIT BUFFER EMPTY
000F 0299                CLR     TI              ;WHEN EMPTY, CLEAR FLAG AND
0011 F599                MOV     SBUF,A          ;SEND CHARACTER
0013 22                  RET

                ;*************************************************************
                ;INTPUT CHARACTER TO ACC
                ;*************************************************************
0014 3098F0     INCHAR:  JNB     RI,$            ;WAIT FOR RECEIVE BUFFER FULL
0017 C298                CLR     RI              ;WHEN CHAR ARRIVES, CLEAR FLAG
0019 E599                MOV     A,SBUF          ;& INPUT CHAR TO ACC
001B 22                  RET

                ;*************************************************************
                ; OUTPUT NULL-TERMINATED STRING
                ;*************************************************************
001C E4         OUTSTR:  CLR     A               ;DPTR POINTS TO STRING OF CHAR
001D 93                  MOVC    A,@A+DPTR       ;GET CHARACTER
001E 6006                JZ      EXIT            ;IF NULL BYTE,DONE
0020 120000              CALL    OUTCHR          ;OTHERWISE, SEND IT
0023 A3                  INC     DPTR            ;POINT TO NEXT CHARACTER
0024 80F6                JMP     OUTSTR          ;AND SEND IT TOO
0026 22         EXIT:    RET

                ;*************************************************************
                ;INPUT CHARACTERS TO BUFFER
                ;*************************************************************
0027 7800   F   INLINE:  MOV     R0,#BUFFER      ;USE R0 AS POINTER TO BUFFER
0029 120000 F   AGAIN:   CALL    INCHAR          ;GET A CHARACTER
002C 120000 F            CALL    OUTCHR          ;ECHO IT BACK
002F F6                  MOV     @R0,A           ;PUT IT IN BUFFER
0030 08                  INC     R0              ;INCRE. POINTER TO BUFFER
0031 B40DF5              CJNE    A,#CR,AGAIN     ;IF NOT CR, GET ANOTHER
                                                 ;CHAR
0034 7600                MOV     @R0,#0          ;PUT NULL BYTE AT END
0036 22                  RET
```

Hình 7.7 : ( tiếp theo )

```
                      ;******************************************************************
                      ;OUTPUT CONTENTS OF BUFFER
                      ;******************************************************************
0037 7800     F       OUTLINE: MOV    R0,#BUFFER  ;USE R0 AS POINTER TO BUFFER
0039 E6               AGAIN2:  MOV    A,@R0       ;GET CHARACTER FROM BUFFER
003A 6006                      JZ     EXIT2       ;IF NULL BYTE,DONE
003C 120000   F                CALL   OUTCHR      ;OTHERWISE, SEND IT
003F 08                        INC    R0          ;POINT TO NEXT CHAR IN BUFFER
0040 80F7                      JMP    AGAIN2      ; AND SEND IT TOO
0042 22               EXIT2:   RET

                      ;******************************************************************
                      ; CREATE A BUFFER IN ONCHIP RAM
                      ;******************************************************************
----                           RSEG   ONCHIP   ;BEGIN RELOCATABLE DATA SEG
0000                  BUFFER:  DS     LENGTH   ;ALLOCATE INTERNAL RAM AS BUF
                               END
```

SYMBOL TABLE LISTING

------ ----- -------

| NAME | TYPE | VALUE | | ATTRIBUTES |
|---|---|---|---|---|
| AGAIN . . . | C ADDR | 0029H | R | SEG=EPROM |
| AGAIN2. . . | C ADDR | 0039H | R | SEG=EPROM |
| BUFFER. . . | D ADDR | 0000H | R | SEG=ONCHIP |
| CR. . . . . | NUMB | 000DH | A | |
| EPROM . . . | C SEG | 0043H | | REL=UNIT |
| EXIT. . . . | C ADDR | 0026H | R | SEG=EPROM |
| EXIT2 . . . | C ADDR | 0042H | R | SEG=EPROM |
| INCHAR. . . | C ADDR | 0014H | R PUB | SEG=EPROM |
| INIT. . . . | C ADDR | 0000H | R PUB | SEG=EPROM |
| INLINE. . . | C ADDR | 0027H | R PUB | SEG=EPROM |
| LENGTH. . . | NUMB | 0028H | A | |
| ONCHIP. . . | D SEG | 0028H | | REL=UNIT |
| OUTCHR. . . | C ADDR | 000CH | R PUB | SEG=EPROM |
| OUTLINE . . | C ADDR | 0037H | R PUB | SEG=EPROM |
| OUTSTR. . . | C ADDR | 001CH | R PUB | SEG=EPROM |
| RI. . . . . | B ADDR | 0098H.0 | A | |
| SBUF. . . . | D ADDR | 0099H | A | |
| SCON. . . . | D ADDR | 0098H | A | |
| SUBROUTINES | ---- | ---- | | |
| TH1 . . . . | D ADDR | 008DH | A | |
| TI. . . . . | B ADDR | 0098H.1 | A | |

**Hình 7.7 : ( tiếp theo )**

```
TMOD. . . .          D ADDR      0089H    A
TR1 . . . .          B ADDR      0088H.6  A

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND

DATE : 03/17/91
DOS 3.31 (038-N) MCS-51 RELOCATOR AND LINKER V3.0, INVOKED BY:
C:\ASM51\RL51.EXE ECHO.OBJ, IO.OBJ TO EXAMPLE CODE(EPROM(8000H))DATA(ONCHIP(30H
>>))

INPUT MODULES INCLUDED
  ECHO.OBJ(MAIN)
  IO.OBJ(SUBROUTINES)

LINK MAP FOR EXAMPLE(MAIN)

        TYPE     BASE     LENGTH     RELOCATION     SEGMENT NAME
        - - - -  - - - - -  - - - - - - -  - - - - - - - - - - -  - - - - - - - - - - - - -

        REG      0000H    0008H                     "REG BANK O"
                 0008H    0028H                     *** GAP ***
        DATA     0030H    0028H      UNIT           ONCHIP

                 0000H    8000H                     *** GAP ***
        CODE     8000H    0067H      UNIT           EPROM

SYMBOL TABLE FOR EXAMPLE(MAIN)

VALUE          TYPE         NAME
- - - - - -    - - - -      - - - - -

- - - - - - -  MODULE       MAIN
N:000DH        SYMBOL       CR
C:8000H        SEGMENT      EPROM
C:8003H        SYMBOL       LOOP
C:8000H        SYMBOL       MAIN
C:8011H        SYMBOL       PROMPT
- - - - - -    ENDMOD       MAIN
- - - - - -    MODULE       SUBROUTINES
C:804DH        SYMBOL       AGAIN
```

Hình 7.7 : ( tiếp theo )

| | | |
|---|---|---|
| C:805DH | SYMBOL | AGAIN2 |
| D:0030H | SYMBOL | BUFFER |
| N:000DH | SYMBOL | CR |
| C:8000H | SEGMENT | EPROM |
| C:804AH | SYMBOL | EXIT |
| C:8066H | SYMBOL | EXIT2 |
| C:8038H | PUBLIC | INCHAR |
| C:8024H | PUBLIC | INIT |
| C:804BH | PUBLIC | INLINE |
| N:0028H | SYMBOL | LENGTH |
| D:0030H | SEGMENT | ONCHIP |
| C:8030H | PUBLIC | OUTCHR |
| C:805BH | PUBLIC | OUTLINE |
| C:8040H | PUBLIC | OUTSTR |
| B:0098H | SYMBOL | RI |
| D:0099H | SYMBOL | SBUF |
| D:0098H | SYMBOL | SCON |
| D:008DH | SYMBOL | TH1 |
| B:0098H.1 | SYMBOL | TI |
| D:0089H | SYMBOL | TMOD |
| B:0088H.6 | SYMBOL | TR1 |
| - - - - - - | ENDMOD | SUBROUTINES |

Hình 7.7 : ( tiếp theo )

Chương trình thực hiện các công việc sau :

- khởi động *port* nối tiếp ( 1 lần )
- xuất dòng chữ " Enter a command "
- nhập một dòng từ bàn phím, hiển thị từng ký tự nhận được
- hiển thị lại cả dòng
- lặp lại

Hình 7.7 trình bày :

(a) tập tin liệt kê ( ECHO.LST ) cho tập tin nguồn đầu tiên

(b) tập tin liệt kê ( IO.LST ) cho tập tin nguồn thứ hai

(c) tập tin liệt kê ( EXAMPLE.M51 ) được tạo ra bởi trình liên kết và định vị.

### 7.8.1 ECHO.LST

Hình 7.7 trình bày nội dung của tập tin ECHO.LST được tạo ra bởi trình dịch hợp ngữ ASM51 khi tập tin nguồn ECHO.SRC được hợp dịch. Các dòng đầu tiên trong tập tin liệt kê này cung cấp cho ta các thông tin tổng quát về môi trường lập trình. Trong số các thông tin này có dòng lệnh mà ta yêu cầu trình dịch hợp ngữ ( ở dạng mở rộng với đường dẫn cho tập tin nguồn ). Lưu ý là ta đã sử dụng điều khiển EP ( ERROR-PRINT ) của trình dịch hợp ngữ trên dòng lệnh và điều này làm cho thông báo lỗi nếu có được gởi ra tập tin liệt kê.

Tập tin nguồn được trình bày bên dưới cột có tiêu đề SOURCE, ngay bên phải của cột LINE. Như ta đã thấy, ECHO.SRC chứa 22 dòng. Dòng 1 đến dòng 4 chứa các điều khiển của trình dịch hợp ngữ.

√$DEBUG ở dòng 1 ra lệnh cho ASM51 đặt bảng ký hiệu vào trong tập tin đối tượng ECHO.OBJ. Điều này cần thiết để trình liên kết và định vị tạo ra một bảng ký hiệu trong tập tin liệt kê.

√$TITLE định nghĩa một chuỗi được đặt ở đầu mỗi trang trong tập tin liệt kê.

√$PAGEWIDTH xác định kích thước cực đại của mỗi dòng trong tập tin liệt kê.

√$NOPAGING tránh không xảy ra bỏ trang đang được chèn vào tập tin liệt kê. Hầu hết các điều khiển của các trình dịch hợp ngữ ảnh hưởng lên việc trình bày tập tin liệt kê. Một vài thử-và-sửa cũng thường tạo ra kết quả theo yêu cầu trong việc in ấn.

Chỉ dẫn NAME ở dòng 6 xác định tập tin hiện hành như là một phần của mô-đun MAIN. Với thí dụ này ta không có thể hiện nào nữa của mô-đun MAIN được sử dụng đến, tuy nhiên các dự án lớn có thể bao gồm các tập tin khác cũng được định nghĩa như là một phần của mô-đun MAIN.

Dòng 7 và dòng 8 nhận dạng các ký hiệu được sử dụng trong mô-đun hiện hành nhưng lại được định nghĩa ở chỗ khác. Nếu không có các chỉ dẫn EXTRN, ASM51 sẽ tạo ra thông báo " undefined symbol " trên mỗi một dòng trong tập tin nguồn mà ở những dòng này có các ký hiệu vừa nêu được sử dụng.

Loại *segment* ( segment type ) cũng phải được xác định cho mỗi một ký hiệu để đảm bảo sử dụng chúng đúng. Tất cả các ký hiệu ngoài được định nghĩa trong thí dụ này thuộc loại CODE.

Bây giờ đến các định nghĩa ký hiệu. Dòng 10 định nghĩa ký hiệu CR là mã ASCII 0DH của ký tự xuống dòng. Dòng 11 định nghĩa ký hiệu

EPROM là *segment* thuộc loại CODE. Nhắc lại là chỉ dẫn SEGMENT chỉ định nghĩa ký hiệu là gì, không có gì khác nữa.

Chỉ dẫn RSEG ở dòng 13 bắt đầu *segment* tái định vị có tên là EPROM. Các lệnh, các hằng dữ liệu và v.v... tiếp theo sẽ được đặt trong *segment* chương trình EPROM.

Chương trình bắt đầu ở dòng 14 tại nhãn MAIN. Lệnh đầu tiên trong chương trình này là một lời gọi đến chương trình con INIT, chương trình khởi động *port* nối tiếp của 8051. Các mã được hợp dịch ở dưới cột OBJ chứa opcode ( 12H cho lệnh LCALL ), tuy nhiên, byte 2 và byte 3 của lệnh ( địa chỉ của chương trình con ) lại xuất hiện là 0000H và được theo sau bởi ký tự F.

Trình liên kết và định vị phải xác định địa chỉ của chương trình con khi các mô-đun chương trình được liên kết với nhau và các địa chỉ được thiết lập cho các *segment* tái định vị. Ta cũng chú ý là địa chỉ dưới cột LOC cũng được cho bằng 0000H.

Do *segment* EPROM thuộc loại tái định vị, trong thời gian dịch ta không biết được nơi *segment* sẽ bắt đầu. Tất cả các *segment* tái định vị sẽ được gán địa chỉ bắt đầu là 0000H trong tập tin liệt kê.

Phần còn lại của chương trình là các lệnh từ dòng 15 đến 19. Một thông báo nhắc được gởi đến VDT bằng cách nạp cho con trỏ dữ liệu DPTR địa chỉ bắt đầu của thông báo và gọi chương trình con OUTSTR. Vì các chương trình con OUTSTR, INLINE và OUTLINE không được định nghĩa trong ECHO.SRC, ta chỉ có thể dự đoán hoạt động của chúng từ tên của chương trình con và các dòng chú thích.

Thông báo nhắc là một chuỗi mã ASCII được kết thúc bằng ký tự NULL, thông báo này được đặt trong *segment* mã EPROM bằng cách sử dụng chỉ dẫn định nghĩa DB ở dòng 21. Vì số byte của thông báo nhắc là hằng số ( nghĩa là không đổi ),ta có thể đặt chính xác chúng trong bộ nhớ chương trình ( dù rằng chúng là các byte dữ liệu ). Thông báo nhắc bắt đầu bằng ký tự xuống dòng để đảm bảo rằng thông báo được hiển thị trên dòng mới. Trong thí dụ này ta giả sử VDT biến đổi CR thành CR/LF.

Tất cả các ký hiệu và các nhãn trong ECHO.SRC đều xuất hiện trong bảng ký hiệu ở phía dưới tập tin ECHO.LST. Vì *segment* EPROM thuộc loại tái định vị và các chương trình con ở bên ngoài, cột VALUE không được dùng đến. Giá trị của ký hiệu EPROM ở cột VALUE cho ta chiều dài của *segment*, trong trường hợp này là 24H hoặc 36 byte.

### 7.8.2 IO.LST

Hình 7.7 cũng trình bày nội dung của tập tin IO.LST – tập tin này chứa các chương trình con xuất nhập. Mô-đun này được gọi tên là SUB-ROUTINES ở dòng 6. Các dòng 7 và 8 khai báo tất cả các tên của các chương trình con là các ký hiệu PUBLIC. Điều này làm cho các ký hiệu được sử dụng bởi các mô-đun khác.

Lưu ý là mọi ký hiệu ( tên của các chương trình con ) đều được khai báo bằng chỉ dẫn PUBLIC mặc dù chỉ có 4 trong chúng được sử dụng trong mô-đun MAIN. Có lẽ khi chương trình phát triển, các mô-đun khác sẽ được thêm vào và chúng sẽ cần đến các chương trình con này.

Các dòng từ 13 đến 16 định nghĩa vài ký hiệu. Một lần nữa EPROM lại được sử dụng làm tên của *segment* mã. Một *segment* khác được dùng trong mô-đun này, ONCHIP. ONCHIP được định nghĩa ở dòng 17 là *segment* dữ liệu nội.

Đến lượt các chương trình con được bắt đầu từ dòng 20. Khối chú thích bắt đầu cho mỗi chương trình con được ghi vắn tắt trong thí dụ này, tuy nhiên ta có thể mô tả chi tiết hơn cho từng chương trình con và điều này thường được dùng đến. Thông thường ta cần cung cấp các điều kiện nhập và thoát cho từng chương trình con.

Sau *segment* cuối cùng này, một vùng đệm trong RAM nội được tạo ra bằng cách dùng *segment* ONCHIP. *Segment* này được bắt đầu bằng cách sử dụng chỉ dẫn RSEG ( dòng 82 ) và vùng đệm được tạo ra bằng cách sử dụng chỉ dẫn DS ( dòng 83 ). Chiều dài của vùng đệm là 40 và được gán cho ký hiệu LENGTH ở đầu của chương trình ( dòng 14 ).

Việc đặt trong tập tin nguồn định nghĩa của ký hiệu LENGTH và thể hiện của *segment* ONCHIP là một vấn đề có ý nghĩa lớn. Cả hai cũng có thể được đặt ngay trước hoặc sau chương trình con INLINE, nơi chúng được sử dụng.

Cũng như *segment* EPROM, *segment* ONCHIP được gán địa chỉ bắt đầu là 0000H dưới cột LOC ở dòng 83. Một lần nữa vị trí thực sự của *segment* ONCHIP sẽ không được xác định cho đến thời gian liên kết ( xem phía dưới ) và ký tự F xuất hiện ở nhiều vị trí trong IO.LST. Mỗi một dòng tương ứng chứa một lệnh sử dụng một ký hiệu mà giá trị chưa được xác định ở thời gian dịch cũng được nhận dạng.

Các zero đặt trong tập tin đối tượng ở các vị trí này sẽ được thay thế bằng các giá trị tuyệt đối bởi trình liên kết và định vị.

### 7.8.3 EXAMPLE.M51

Hình 7.7 còn trình bày nội dung của tập tin EXAMPLE.M51 được tạo ra bởi trình liên kết và định vị, RL51. Dòng lệnh yêu cầu RL51 được lặp lại ở phía trên của EXAMPLE.M51 và ta nên khảo sát cẩn thận.

Dưới đây là dòng lệnh này ( bỏ qua phần đường dẫn ) :

```
RL51 ECHO.OBJ, IO.OBJ TO EXAMPLE CODE ( EPROM (8000H ) ) &
DATA ( ONCHIP ( 30H ) )
```

Tiếp theo lệnh điều khiển ( command ) RL51, các mô-đun đối tượng được liệt kê cách nhau bởi dấu phẩy theo trật tự chúng được liên kết. Tiếp theo danh sách các mô-đun, điều khiển tùy chọn TO EXAMPLE cung cấp tên của mô-đun đối tượng tuyệt đối được tạo ra bởi RL51. Nếu tùy chọn này bị bỏ qua, tên của mô-đun đối tượng tuyệt đối là tên của tập tin đầu tiên của danh sách các mô-đun ( không có phần mở rộng )

Tập tin liệt kê trong thí dụ này tự động được đặt tên là EXAMPLE.M51. Cuối cùng các điều khiển định vị CODE và DATA xác định tên của các *segment* thuộc loại được kết hợp và địa chỉ tuyệt đối ở đó *segment* được bắt đầu. Trong thí dụ này *segment* mã EPROM bắt đầu ở địa chỉ 8000H còn *segment* dữ liệu ONCHIP bắt đầu ở địa chỉ byte 30H trong RAM nội của 8051.

Tiếp theo dòng lệnh yêu cầu RL51 được liệt kê lại, EXAMPLE.M51 chứa một danh sách các mô-đun được liên kết bởi RL51. Trong thí dụ này ta chỉ có hai tập tin ( ECHO.OBJ và IO.OBJ ) và hai mô-đun ( MAIN và SUBROUTINES ) được liệt kê. Nếu chỉ dẫn NAME không được sử dụng trong các tập tin nguồn, tên của các mô-đun sẽ giống như tên của các tập tin.

Bản đồ liên kết xuất hiện tiếp theo trong tập tin liệt kê. Cả hai *segment* EPROM và ONCHIP được nhận dạng cùng với địa chỉ bắt đầu và chiều dài của chúng. ONCHIP được nhận dạng là *segment* dữ liệu bắt đầu ở địa chỉ 30H và có chiều dài là 40 byte ( 28H ). EPROM được nhận dạng là là *segment* mã bắt đầu ở địa chỉ 8000H và có chiều dài là 103 byte ( 67H ).

Cuối cùng, EXAMPLE.M51 chứa một bảng ký hiệu. Tất cả các ký hiệu ( bao gồm cả các nhãn ) sử dụng trong chương trình đều được liệt kê, được sắp thứ tự dựa trên mô-đun. Tất cả các giá trị là tuyệt đối. Nên nhớ rằng bảng ký hiệu trong tập tin .M51 chỉ có thể được tạo ra nếu điều khiển $DEBUG của trình dịch hợp ngữ được đặt ở đầu mỗi một tập tin nguồn. Địa chỉ của chương trình con INIT ( địa chỉ vắng mặt trong ECHO.LST mà ta đã lưu ý trước đây ) được nhận dạng dưới mô-đun SUBROUTINES là 8024H. Địa chỉ này được thay thế bằng địa chỉ mã

trong mô-đun đối tượng nào đó có sử dụng lệnh CALL INIT, đã được lưu ý trước đây trong mô-đun MAIN. Việc biết giá trị tuyệt đối của các nhãn rất quan trọng khi gỡ rối chương trình. Khi một lỗi được tìm thấy, thông thường một mảng chương trình tạm thời có thể được thực hiện bằng cách sửa đổi các byte của chương trình và thực thi chương trình. Nếu mảng chương trình này có lỗi, việc sửa đổi được thực hiện ở chương trình nguồn.

### 7.9 *MACRO*

Đây là vấn đề cuối cùng của chương này và ta quay trở lại ASM51. Tiện ích xử lý *macro* MPL của ASM51 là tiện ích thay thế chuỗi ( string replacement facility ). Các *macro* cho phép sử dụng các phần của chương trình, các phần này chỉ cần được định nghĩa một lần bằng cách sử dụng một mã gợi nhớ đơn giản và được sử dụng ở bất kỳ nơi nào trong chương trình bằng cách chèn mã gợi nhớ vào. Việc lập trình sử dụng *macro* là một mở rộng của các kỹ thuật lập trình đã được mô tả trước đây. *Macro* có thể được định nghĩa ở bất kỳ nơi nào trong chương trình nguồn và được sử dụng giống như một lệnh. Cú pháp cho định nghĩa *macro* là :

```
%*DEFINE    (call_pattern) ( macro_body )
```

Call_pattern : biểu đồ gọi

Macro_body : thân *macro*

Một khi được định nghĩa, biểu đồ gọi giống như một mã gợi nhớ; biểu đồ có thể được sử dụng giống như một lệnh của hợp ngữ bằng cách đặt biểu đồ vào trong trường mã gợi nhớ của lệnh. Các *macro* được phân biệt với các lệnh thật sự của hợp ngữ bằng cách đặt trước chúng một dấu " % ". Khi chương trình nguồn được hợp dịch, mọi thứ trong thân *macro* được thay thế bằng biểu đồ gọi dựa trên việc ánh xạ 1-1 từng ký tự. Các *macro* cung cấp một phương tiện đơn giản để thay thế các biểu đồ lệnh phức tạp bởi các mã gợi nhớ đơn giản, dễ nhớ. Cần nhắc lại là việc thay thế dựa trên cơ sở ánh xạ 1-1 từng ký tự, không có gì thêm cả.

Thí dụ nếu định nghĩa *macro* sau đây xuất hiện ở nơi bắt đầu của tập tin nguồn :

```
%*DEFINE    ( PUSH_DPTR )
            ( PUSH  DPH
            PUSH  DPL
            )
```

thì phát biểu

```
%PUSH_DPTR
```

sẽ xuất hiện trong tập tin liệt kê như sau :

```
PUSH   DPH
PUSH   DPL
```

Thí dụ trên là một *macro* điển hình. Vì các lệnh liên quan đến *stack* của 8051 chỉ hoạt động với các địa chỉ trực tiếp, việc cất con trỏ dữ liệu yêu cầu hai lệnh PUSH. Một *macro* tương tự có thể được tạo ra cho việc POP con trỏ dữ liệu.

Dưới đây là một vài lợi ích khi ta sử dụng *macro* :

- Chương trình nguồn sử dụng *macro* sẽ dễ đọc hơn vì mã gợi nhớ của *macro*, một cách tổng quát, cho ta thấy rõ hoạt động được dự định hơn là các lệnh tương đương của hợp ngữ.
- Chương trình nguồn ngắn hơn.
- Việc sử dụng *macro* sẽ giảm các lỗi.
- Việc sử dụng *macro* giải phóng người lập trình khỏi việc xử lý các chi tiết cấp thấp.

Hai lợi ích sau cùng nêu ở trên có liên quan với nhau. Một khi *macro* đã được viết và được gỡ rối, *macro* được sử dụng mà không phải lo lắng về các lỗi. Trong thí dụ PUSH_DPTR ở trên, nếu các lệnh PUSH và POP được sử dụng thay vì là các *macro* cất vào *stack* và lấy ra từ *stack*, người lập trình có thể vô ý đảo ngược thứ tự của DPL và DPH trong khi sử dụng các lệnh PUSH và POP ( byte cao hay byte thấp được cất vào *stack* trước tiên ? ). Điều này sẽ sinh ra lỗi. Tuy nhiên, bằng việc sử dụng *macro*, các chi tiết trên chỉ cần được quan tâm một lần khi *macro* được viết, sau đó ta tự do sử dụng *macro* mà không lo lắng về các lỗi.

Vì việc thay thế dựa trên cơ sở ánh xạ 1-1 từng ký tự, định nghĩa *macro* cần được cấu trúc một cách cẩn thận với các ký tự xuống dòng , tab, v.v... để đảm bảo việc gán đúng các phát biểu của *macro* với phần còn lại của chương trình hợp ngữ.

Có nhiều đặc trưng cải tiến công cụ xử lý *macro* của ASM51 cho phép ta truyền tham số, các nhãn cục bộ, các hoạt động lặp lại, điều khiển luồng và v.v...

### 7.9.1 Truyền tham số

Một *macro* có các tham số được truyền từ chương trình gọi có dạng được sửa đổi như sau :

```
%*DEFINE   ( macro_name ( parameter_list ) ) ( macro_body )
```

macro_name : tên của *macro*

parameter_list : danh sách tham số

macro_body : thân *macro*

Thí dụ nếu *macro* sau được định nghĩa :

```
%DEFINE     (CMPA#  (VALUE))
            (CJNE   A, #%VALUE, $ + 3
            )
```

thì lời gọi *macro*

```
%CMPA# (20H)
```

được mở rộng thành lệnh sau trong tập tin .LST :

```
CJNE   A, #20H, $ + 3
```

Mặc dù 8051 không có lệnh so sánh thanh chứa, ta có thể tạo ra dễ dàng bằng cách sử dụng lệnh CJNE với " $ + 3 " ( lệnh kế ) làm đích cho nhảy có điều kiện. Mã gợi nhớ CMPA# có thể được nhớ dễ dàng hơn đối với nhiều người lập trình. Việc sử dụng *macro* còn giảm gánh nặng cho người lập trình khỏi phải nhớ các chi tiết có tính chú thích như là " $ + 3 " chẳng hạn.

Chúng ta hãy mở rộng thí dụ trên. Thật là tốt nếu 8051 có các lệnh sau :

JUMP IF ACCUMULATOR GREATER THAN X

JUMP IF ACCUMULATOR GREATER THAN OR EQUAL TO X

JUMP IF ACCUMULATOR LESS THAN X

JUMP IF ACCUMULATOR LESS THAN OR EQUAL TO X

nhưng *chip* vi điều khiển này lại không có. Các thao tác trên có thể được tạo ra bằng cách sử dụng lệnh CJNE sau đó là lệnh JC hoặc JNC, nhưng những chi tiết này lại phức tạp. Thí dụ giả sử chương trình cần nhảy đến nhãn GREATER_THAN nếu thanh chứa chứa một mã ASCII lớn hơn " Z " ( 5AH ). Chuỗi lệnh sau đây có thể sử dụng :

```
CJNE   A, #5BH, $ + 3
JNC    GREATER_THAN
```

Lệnh CJNE trừ bớt 5BH ( nghĩa là " Z " + 1 ) cho nội dung thanh chứa và *set* bằng 1 hoặc xóa bằng 0 cờ nhớ. Lệnh CJNE làm cho C = 1 với các giá trị của thanh chứa từ 00H trở lên kể cả 5AH. Lưu ý là 5AH – 5BH < 0 do vậy C = 1 nhưng 5BH – 5BH = 0 nên C = 0. Việc nhảy đến GREATER_THAN dựa trên điều kiện " không có nhớ " sẽ được thực hiện

nếu nội dung của thanh chứa là các giá trị 5BH, 5CH, 5DH, v.v... cho đến FFH và ngược lại việc nhảy không được thực hiện nếu thanh chứa chứa các giá trị từ 00H đến 5AH. Công việc sẽ đơn giản hơn nếu ta tạo ra một mã gợi nhớ thích hợp cho một định nghĩa *macro*, và sử dụng *macro* này thay vì chuỗi lệnh tương ứng.

Dưới đây là định nghĩa cho *macro* " nhảy nếu lớn hơn " :

```
%*DEFINE      ( JGT ( VALUE, LABEL ) )
              ( CJNE    A, #%VALUE + 1, $ + 3
              JNC     %LABEL
              )
```

Để kiểm tra xem có phải thanh chứa A chứa một mã ASCII lớn hơn " Z " hay không, ta gọi *macro* như sau :

```
%JGT ( 'Z', GREATER_THAN )
```

ASM51 mở rộng *macro* như sau :

```
CJNE   A, #5BH, $ + 3
JNC    GREATER_THAN
```

*Macro* JGT là một thí dụ tốt cho việc sử dụng *macro*. Bằng cách sử dụng *macro* này, người lập trình tránh được các chi tiết rắc rối và dễ tạo ra lỗi.

### 7.9.2 Các nhãn cục bộ

Các nhãn cục bộ có thể được sử dụng trong một *macro* bằng cách dùng khuôn dạng sau :

```
%DEFINE ( macro_name [ ( parameter_list ) ] )
              [ LOCAL  list_of_local_labels ] ( macro_body )
```

macro_name : tên của *macro*

parameter_list : danh sách tham số

macro_body : thân *macro*

list_of_local_labels : danh sách các nhãn cục bộ

Thí dụ ta có định nghĩa *macro* sau :

```
%DEFINE ( DEC_DPTR ) LOCAL  SKIP
           ( DEC   DPL
           MOV   A, DPL
           CJNE  A, #0FFH, %SKIP
           DEC   DPH
%SKIP:   )
```

*macro* nên được gọi như sau :

```
    %DEC_DPTR
```

và được ASM51 mở rộng thành :

```
        DEC     DPL
        MOV     A, DPL
        CJNE    A, #0FFH, SKIP00
        DEC     DPH
SKIP00:
```

Lưu ý là một nhãn cục bộ trong trường hợp tổng quát không xung đột với nhãn cùng tên được sử dụng ở nơi khác trong chương trình nguồn, do ASM51 nối thêm một mã số cho nhãn cục bộ khi *macro* được mở rộng. Hơn nữa, nếu sử dụng tiếp theo nhãn cục bộ vừa nêu, ASM51 lại nối thêm một mã số khác cho nhãn.

Trong thí dụ *macro* trên, thanh chứa A được sử dụng để lưu giữ tạm thời nội dung của DPL. Nếu *macro* này được gọi đến ở trong một phần của chương trình mà phần này lại sử dụng thanh chứa A cho mục đích khác, giá trị của A sẽ bị mất. Điều này gây ra lỗi trong chương trình. Ta có thể tránh được lỗi này bằng cách cất nội dung thanh chứa A vào *stack* sau đó phục hồi lại ở cuối *macro*. Dưới đây là định nghĩa cho *macro* vừa đề cập sau khi sửa đổi :

```
%*DEFINE ( DEC_DPTR ) LOCAL SKIP
            ( PUSH  ACC
            DEC     DPL
            MOV     A, DPL
            CJNE    A, #0FFH, %SKIP
            DEC     DPH
%SKIP: POP  ACC
            )
```

### 7.9.3 Thao tác lặp lại

Đây là một trong vài *macro* được cài đặt sẵn ( định nghĩa trước ). Dạng của *macro* này là :

```
    %REPEAT     ( expression ) ( text )
```

**expression : biểu thức**

**text : văn bản**

Lấy thí dụ để làm đầy một khối trong bộ nhớ bằng 100 lệnh NOP, ta sử dụng *macro* như sau :

```
%REPEAT ( 100 )
        ( NOP
        )
```

### 7.9.4 Thao tác luồng điều khiển

Việc hợp dịch có điều kiện các phần của chương trình được thực hiện bằng cách sử dụng định nghĩa *macro* luồng điều khiển của ASM51. Dạng của *macro* này như sau :

```
%IF ( expression ) THEN ( balanced_text )
( ELSE ( balanced_text ) )
```

Thí dụ :

```
INTERNAL    EQU    1                ; 1 = 8051 serial I/O drivers
                                    ; 0 = 8251 serial I/O drivers
            .
            .
            %IF ( INTERNAL ) THEN
( INCHAR :  .                       ; 8051 drivers
            .
OUTCHR:     .
            .
            ) ELSE
( INCHAR:   .                       ; 8251 drivers
            .
OUTCHR:     .
            .
            )
```

Trong thí dụ trên, ký hiệu INTERNAL được gán giá trị 1 để chọn các chương trình con xuất nhập cho *port* nối tiếp của 8051 hoặc giá trị 0 để chọn các chương trình con xuất nhập cho UART bên ngoài, trong trường hợp này là 8251. *Macro* IF làm cho ASM51 dịch tập các chương trình con này và bỏ qua tập các chương trình con khác.

Các chương trình con INCHAR và OUTCHR được sử dụng ở bất kỳ nơi nào trong chương trình mà không cần khảo sát cấu hình phần cứng cụ thể. Chỉ cần chương trình được dịch với giá trị đúng của INTERNAL, chương trình con tương ứng sẽ được thực thi.

# 8

# CẤU TRÚC CHƯƠNG TRÌNH

## 1.1 MỞ ĐẦU

Chương này giới thiệu các đặc trưng của một chương trình và một vài kỹ thuật dùng để phát triển một chương trình. Ta sẽ bắt đầu bằng việc giới thiệu các kỹ thuật lập trình có cấu trúc.

Lập trình có cấu trúc ( structured programming ) là một kỹ thuật dùng để tổ chức và viết chương trình sao cho giảm được độ phức tạp, cải thiện tính rõ ràng của chương trình đồng thời dễ dàng gỡ rối, sửa chữa cũng như thay đổi. Ý tưởng về các chương trình được cấu trúc một cách rõ ràng thường được nhấn mạnh trong hầu hết các công việc lập trình và ở đây ta thúc đẩy ý tưởng này. Khả năng của phương pháp này có thể được đánh giá bằng cách khảo sát phát biểu sau : mọi chương trình đều có thể được viết bằng cách chỉ sử dụng ba cấu trúc : " các phát biểu ", " các vòng lặp " và " các lựa chọn ".

" Các phát biểu ", " các vòng lặp ", " các lựa chọn " tạo thành một tập đầy đủ các cấu trúc và mọi chương trình đều có thể được hiện thực bằng cách chỉ sử dụng ba cấu trúc này. Điều khiển chương trình được chuyển qua một cấu trúc đến cấu trúc khác mà không có các rẽ nhánh không điều kiện. Mỗi một cấu trúc có một điểm nhập và một điểm thoát.

Một cách điển hình, một chương trình có cấu trúc chứa một trật tự có thứ bậc các chương trình con, mỗi một chương trình con có một điểm nhập và một điểm thoát.

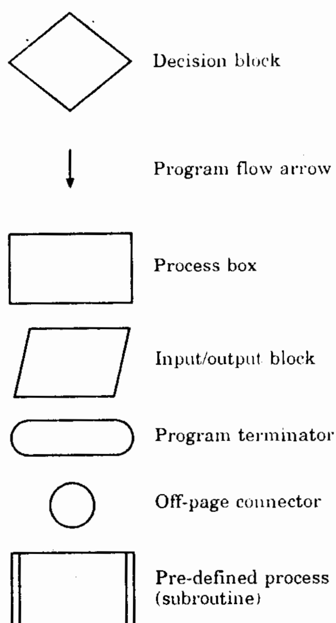
Mục đích của chương này là giới thiệu việc lập trình có cấu trúc áp dụng cho lập trình hợp ngữ. Mặc dù các ngôn ngữ cấp cao ( như Pascal, C, v.v... ) đẩy mạnh việc lập trình có cấu trúc thông qua các phát biểu của chúng ( WHILE, FOR v.v... ) và các quy ước ký hiệu trong khi hợp ngữ không có các đặc trưng vốn có như vậy, việc sử dụng các kỹ thuật có cấu trúc cũng giúp cho việc lập trình hợp ngữ đạt được những thuận lợi rất lớn.

Để tiến đến mục đích – tạo ra các chương trình hợp ngữ tốt, các bài tập làm thí dụ được thể hiện bằng ba phương pháp :

- lưu đồ ( flowchart )
- giả mã ( pseudo code )
- hợp ngữ ( assembly language )

Việc giải các bài tập lập trình hợp ngữ dĩ nhiên là mục tiêu cuối cùng của chương này, tuy nhiên lưu đồ và giả mã là các công cụ thường dùng cho các giai đoạn khởi đầu. Cả hai, lưu đồ và giả mã đều là những công cụ trực quan giúp ta dễ dàng trình bày và hiểu các bài tập một cách có hệ thống. Cả hai cho phép một bài tập được mô tả dưới dạng " điều gì phải được thực hiện " hơn là " thực hiện bằng cách nào ". Thông thường lời giải có thể được biểu diễn dưới dạng các lưu đồ hoặc giả mã ở dạng các thuật ngữ độc lập với máy, không cần khảo sát những rắc rối khó hiểu của tập lệnh máy.

Có thể ta không sử dụng cả hai phương pháp : giả mã và lưu đồ. Việc lựa chọn phương pháp nào cho thích hợp tùy thuộc vào cá nhân của người lập trình. Các ký hiệu thường dùng nhất cho việc lập lưu đồ được trình bày ở hình 8.1.




Hình 8.1 : Các ký hiệu thường dùng nhất cho việc lập lưu đồ

Decision block : khối quyết định

Program flow arrow : mũi tên chỉ đường đi của chương trình

Process box : hộp xử lý

Input / output block : khối xuất / nhập

Program terminator : kết thúc chương trình

Page off connector : kết nối qua trang

Pre-defined process ( subroutine ) : xử lý tiền định nghĩa ( chương trình con )

Ta có thể xem giả mã như là một loại ngôn ngữ của máy tính. Ý tưởng này đã được sử dụng không chính thức trong quá khứ như là một phương pháp thuận lợi để phát thảo lời giải cho các bài tập lập trình. Khi được áp dụng ở đây, giả mã bắt chước cú pháp của Pascal hoặc C ở dạng ký hiệu cấu trúc, tuy nhiên cũng khuyến khích việc sử dụng ngôn ngữ tự nhiên để mô tả các hoạt động. Vậy thì một phát biểu như là :

```
[ get a character from the keyboard ]
[ nhập một ký tự từ bàn phím ]
```

có thể xuất hiện trong giả mã. Lợi ích của việc sử dụng giả mã là tạo ra sự tôn trọng triệt để đối với cấu trúc đồng thời kết hợp với ngôn ngữ không chính thức. Vậy thì chúng ta có thể thấy :

```
IF [ condition is true ]
    THEN [ do statement 1 ]
    ELSE BEGIN
        [ do statement 2 ]
        [ do statement 3 ]
    END
```

hoặc :

```
IF [ the temperature is less than 20 degrees Celsius ]
    THEN [ wear a jacket ]
    ELSE BEGIN
        [ wear a short sleeve shirt ]
        [ bring sunglasses ]
    END
```

Việc sử dụng các từ khóa ( keyword ), thụt hàng ( indentation ) và lệnh là cần thiết giúp cho việc sử dụng giả mã có hiệu quả. Mục đích của chúng ta là chứng tỏ một cách rõ ràng rằng lời giải cho một bài tập lập trình sử dụng lưu đồ và/hoặc giả mã, sau đó chuyển thành hợp ngữ sẽ dễ dàng hơn so với việc lập trình trực tiếp bằng hợp ngữ. Sản phẩm cuối cùng cũng dễ đọc, dễ gỡ rối và dễ bảo trì hơn.

Giả mã sẽ được định nghĩa một cách chính thức trong mục sau.

## 8.2 ƯU VÀ KHUYẾT ĐIỂM CỦA LẬP TRÌNH CÓ CẤU TRÚC

Các ưu điểm của phương pháp lập trình có cấu trúc có thể được liệt kê như sau :

- dễ dàng vạch ra chuỗi các hoạt động, do vậy thuận tiện cho việc gỡ rối.
- có một số hữu hạn các cấu trúc kèm theo thuật ngữ đã được chuẩn hóa.
- các cấu trúc giúp cho việc xây dựng các chương trình con được dễ dàng.
- tập các cấu trúc là đầy đủ, nghĩa là mọi chương trình đều có thể được viết bằng cách sử dụng ba cấu trúc.
- các cấu trúc giúp ta dễ dàng mô tả bằng lưu đồ, giản đồ cú pháp, giả mã và v.v...
- việc lập trình có cấu trúc dẫn đến kết quả là làm tăng năng suất của người lập trình – chương trình được viết nhanh hơn.

Tuy nhiên việc lập trình có cấu trúc cũng tồn tại một số khuyết điểm sau :

- chỉ có một vài ngôn ngữ cấp cao như Pascal, C, PL/M, v.v... chấp nhận trực tiếp các cấu trúc; các ngôn ngữ cấp cao khác yêu cầu thêm một giai đoạn dịch nữa.
- các chương trình có cấu trúc có thể thực thi chậm hơn và đòi hỏi nhiều bộ nhớ hơn so với một chương trình tương đương nhưng không có cấu trúc.
- một số vấn đề sẽ gặp khó khăn khi giải quyết nếu ta chỉ sử dụng ba cấu trúc để lập trình.
- các cấu trúc lồng vào nhau cũng có khó khăn kèm theo.

## 8.3 BA CẤU TRÚC

Mọi vấn đề lập trình đều có thể được thực hiện bằng cách dùng ba cấu trúc :

- các phát biểu ( statement )
- các vòng lặp ( loop )
- sự lựa chọn ( choice )

Tính đầy đủ của ba cấu trúc dường như không hứa hẹn lắm, tuy nhiên nếu ta thêm vào sự lồng vào nhau ( cấu trúc trong cấu trúc ), ta dễ dàng chứng tỏ rằng một bài tập lập trình bất kỳ đều có thể được thực hiện chỉ với ba cấu trúc này. Ta hãy khảo sát chi tiết từng cấu trúc.

### 8.3.1. Các phát biểu

Các phát biểu cung cấp cho ta cơ chế cơ bản để thực hiện một điều gì đó. Các khả năng bao gồm việc gán một giá trị cho một biến, chẳng hạn như :

```
[ count = 0 ]
```

hoặc gọi một chương trình con, thí dụ :

```
PRINT_STRING ( " Select Option : " )
```

Bất cứ nơi nào mà một phát biểu đơn được sử dụng, ta đều có thể sử dụng một nhóm các phát biểu hay khối phát biểu ( statement block). Điều này có thể được thực hiện với giả mã bằng cách đặt các phát biểu giữa hai từ khóa BEGIN và END như sau :

```
BEGIN
        [ statement 1 ]
        [ statement 2 ]
        [ statement 3 ]
END
```

Lưu ý là các phát biểu trong khối phát biểu được đặt lùi vào so với các từ khóa BEGIN và END, điều này là một đặc trưng quan trọng của lập trình có cấu trúc.

### 8.3.2 Cấu trúc vòng lặp

Cấu trúc cơ bản thứ hai là vòng lặp, cấu trúc này được dùng để lặp lại việc thực hiện một thao tác. Cộng một chuỗi các số hoặc tìm kiếm một giá trị trong một danh sách là hai thí dụ của các bài tập lập trình

đòi hỏi phải có vòng lặp. Thuật ngữ lặp lại ( iteration ) cũng được dùng trong ngữ cảnh này. Mặc dù có vài dạng vòng lặp nhưng ta chỉ cần hai dạng : WHILE / DO và REPEAT / UNTIL.

### 8.3.2.1 Phát biểu WHILE / DO

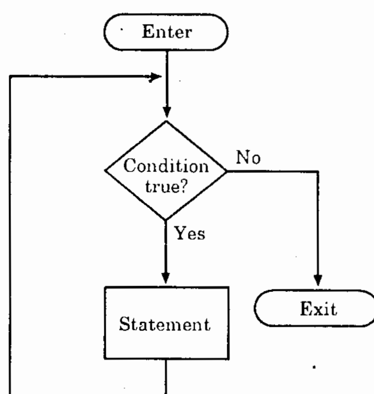
Phát biểu WHILE / DO cho ta phương tiện dễ dàng nhất để thực hiện các vòng lặp. WHILE / DO được gọi là một phát biểu vì WHILE / DO được xử lý như một phát biểu – là đơn vị có một điểm nhập ( bắt đầu ) và một điểm thoát ( kết thúc ). Giả mã có dạng như sau :

```
WHILE [ condition ] DO
        [ statement ]
```

condition : điều kiện

statement : phát biểu

Điều kiện là một biểu thức quan hệ ( ralational expression ) có giá trị là đúng ( true ) hoặc sai ( false ). Nếu điều kiện là đúng ( hay thỏa ), phát biểu ( hay khối phát biểu ) theo sau DO được thực thi và sau đó điều kiện được đánh giá lại. Điều này được lặp lại cho đến khi biểu thức quan hệ có kết quả sai ( hay không thỏa ), phát biểu ( hay khối phát biểu ) được bỏ qua và việc thực thi chương trình sẽ tiếp tục ở phát biểu kế tiếp. Cấu trúc WHILE / DO được trình bày trong lưu đồ hình 8.2.




Hình 8.2 : Lưu đồ của cấu trúc WHILE / DO

Enter : vào ( hay nhập )

Condition true ? : điều kiện đúng ( hay thỏa ) ?

Statement : phát biểu

Exit : thoát

***Thí dụ 8.1*** : Cấu trúc WHILE / DO

*Hãy minh họa cấu trúc WHILE / DO như sau : trong khi cờ nhớ của 8051 còn bằng 1, một phát biểu được thực thi.*

***Lời giải***

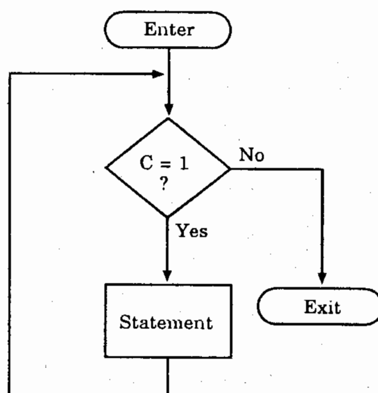
Giả mã :

```
WHILE [ c == 1 ] DO
        [ statement ]
```

statement : phát biểu

( *lưu ý* : dấu bằng kép được dùng để phân biệt toán tử quan hệ ( toán tử này kiểm tra sự bằng nhau ) với toán tử gán chỉ có một dấu bằng ( xem 8.4 : cú pháp của giả mã ).

Lưu đồ : ( hình 8.3 )




Hình 8.3 : Lưu đồ của thí dụ 8.1

Enter : vào ( hay nhập )

Statement : phát biểu

Exit : thoát

Chương trình cho 8051 :

```
ENTER:          JNC    EXIT
STATEMENT:  ( statement )
                JMP    ENTER
EXIT:           ( continue )
```

Statement : phát biểu

Continue : tiếp tục

Qui luật chung là các thao tác trong khối phát biểu phải ảnh hưởng đến ít nhất một biến trong biểu thức quan hệ, ngược lại một lỗi ở dạng một vòng lặp vô tận được tạo ra.

***Thí dụ 8.2*** : Chương trình con SUM

*Hãy viết một chương trình con cho 8051 có tên là SUM, chương trình này tính tổng của một chuỗi số. Các tham số được truyền tới chương trình con bao gồm chiều dài của chuỗi số chứa trong thanh ghi R7 và địa chỉ bắt đầu của chuỗi chứa trong thanh ghi R0 ( giả sử chuỗi hiện diện trong bộ nhớ nội của 8051 ). Kết quả tổng được cất trong thanh chứa A.*

***Lời giải***

Giả mã :

```
[ sum = 0 ]
WHILE  [ length > 0 ]  DO  BEGIN
          [ sum = sum + @pointer ]
          [ increment pointer ]
          [ decrement length ]
END
```

Increment pointer : tăng nội dung con trỏ bởi 1

Decrement length : giảm chiều dài bởi 1

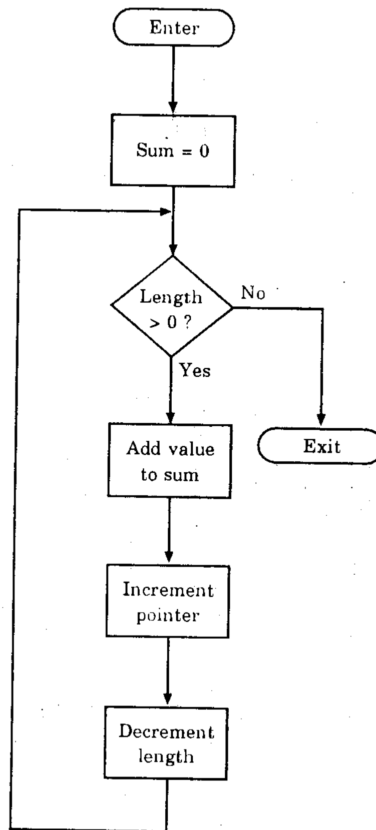
Lưu đồ : ( hình 8.4 )

Chương trình cho 8051 :

```
                ( được cấu trúc chặt : 13 byte )
SUM:            CLR     A
LOOP:           CJNE    R7, #0, STATEMENT
                JMP     EXIT
STATEMENT:      ADD     A, @R0
                INC     R0
                DEC     R7
                JMP     LOOP
EXIT:           RET
```

( được cấu trúc lồng : 9 byte )

```
SUM:      CLR    A
          INC    R7
MORE:     DJNZ   R7, SKIP
          RET
SKIP:     ADD    A, @R0
          INC    R0
          SJMP   MORE
```




Hình 8.4 : Lưu đồ của thí dụ 8.2

Enter : nhập

Add value to sum: cộng giá trị vào tổng

Increment pointer : tăng nội dung con trỏ

Decrement length : giảm chiều dài

Exit : thoát

Ta lưu ý là giải đáp với cấu trúc lỏng 9 byte ngắn hơn và được thực thi nhanh hơn so với giải đáp có cấu trúc chặt 13 byte. Những người lập trình có kinh nghiệm sẽ viết các thí dụ đơn giản như vậy một cách trực giác bằng cấu trúc lỏng ( loose ). Tuy nhiên, những người lập trình tập sự có thể được thuận lợi bằng cách trước tiên giải bài toán một cách rõ ràng bằng giả mã, sau đó tiến hành việc lập trình bằng hợp ngữ theo cấu trúc của giả mã.

***Thí dụ 8.3*** : Cấu trúc WHILE/ DO

*Hãy minh họa cấu trúc WHILE / DO bằng cách sử dụng điều kiện kết hợp sau đây : thanh chứa có nội dung khác 0DH ( carriage return ) và nội dung của thanh ghi R7 không bằng 0.*

***Lời giải***

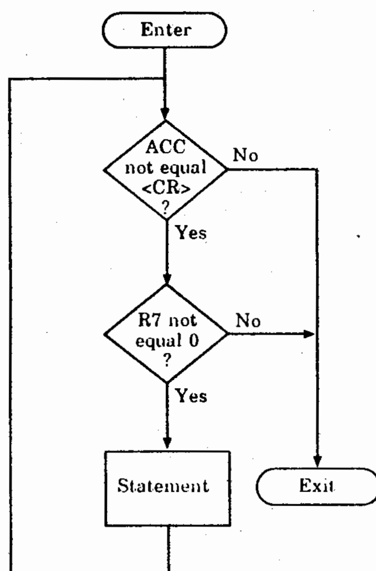
Giả mã :

WHILE [ ACC ! = CR AND R7 ! = 0 ] DO

[ statement ]

**Statement : phát biểu**

Lưu đồ : ( hình 8.5 )




Hình 8.5 : Lưu đồ của thí dụ 8.3

Enter : nhập

ACC not equal <CR> ? : nội dung của ACC khác <CR> ?

R7 not equal 0 ? : nội dung của R7 khác 0 ?

Statement : phát biểu

Exit : thoát

Chương trình cho 8051 :

```
ENTER:        CJNE   A, 0DH, SKIP
              JMP    EXIT
SKIP:         CJNE   R7, #9, STATEMENT
              JMP    EXIT
STATEMENT:    ( one or more statements )
              .
              .
              .
              JMP    ENTER
EXIT          ( continue )
```

One or more statements : một hoặc nhiều phát biểu nữa

Continue : tiếp tục

### 8.3.2.2 Phát biểu REPEAT / UNTIL

Tương tự phát biểu WHILE / DO là phát biểu REPEAT / UNTIL, phát biểu này thường dùng khi việc lặp lại phát biểu phải được thực thi ít nhất một lần.

Phát biểu WHILE / DO trước tiên kiểm tra điều kiện, như vậy có thể phát biểu sẽ không được thực thi nếu điều kiện không thỏa ( cho kết quả sai ).

√ Giả mã :

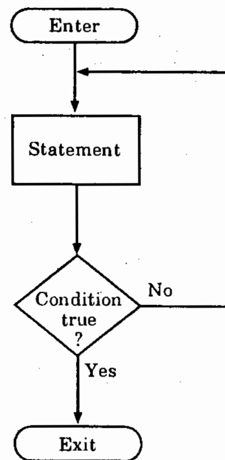
```
REPEAT  [ statement ]
UNTIL   [ condition ]
```

Statement : phát biểu

Condition : điều kiện

Lưu đồ :

( xem hình 8.6 )




Hình 8.6 : Lưu đồ của cấu trúc REPEAT / UNTIL

Enter : nhập

Statement : phát biểu

Condition true ? : điều kiện đúng ?

Exit : thoát

***Thí dụ 8.4*** : Chương trình con tìm kiếm

*Hãy viết một chương trình con cho 8051 để tìm kiếm trong một chuỗi kết thúc bằng NULL ( chuỗi được trỏ bởi R0 ) và xác định có phải ký tự Z hiện diện trong chuỗi không ?.*

*Kết quả trả về ( ACC ) = ' Z ' nếu Z có trong chuỗi và ( ACC ) = 0 trong trường hợp ngược lại.*

***Lời giải***

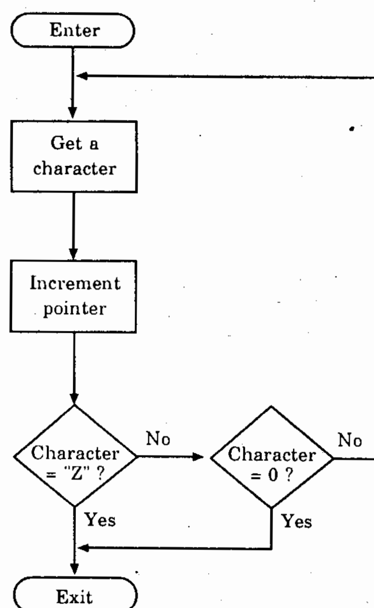
Giả mã :

```
REPEAT
    [ ACC = @pointer ]
    [ increment pointer ]
UNTIL
    [ ACC == 'Z'  or  ACC == 0 ]
```

increment pointer : tăng nội dung con trỏ bởi 1

Lưu đồ :

( xem hình 8.7 )




Hình 8.7 : Lưu đồ của thí dụ 8.4

**Enter : nhập**

**Get a character : Nhập một ký tự**

**Increment pointer : tăng nội dung con trỏ bởi 1**

**Character = ' Z ' ? : ký tự là Z ?**

**Character = 0 ? : ký tự là 0 ?**

**Exit : thoát**

Chương trình cho 8051 :

```
STATEMENT: MOV   A, @R0
           INC   R0
           JZ    EXIT
           CJNE  A, # 'Z', STATEMENT
EXIT:      RET
```

### 8.3.3 Cấu trúc lựa chọn

Cấu trúc cơ bản thứ ba là cấu trúc lựa chọn – " sự rẽ nhánh trên đường " của người lập trình.

Hai loại phát biểu chung nhất của cấu trúc này là phát biểu IF / THEN / ELSE và phát biểu CASE.

### 8.3.3.1 Phát biểu IF / THEN / ELSE

Phát biểu IF / THEN / ELSE được sử dụng khi một trong 2 phát biểu ( hay các khối phát biểu ) phải được lựa chọn, việc lựa chọn phụ thuộc vào một điều kiện. Phần ELSE của phát biểu là tùy chọn.

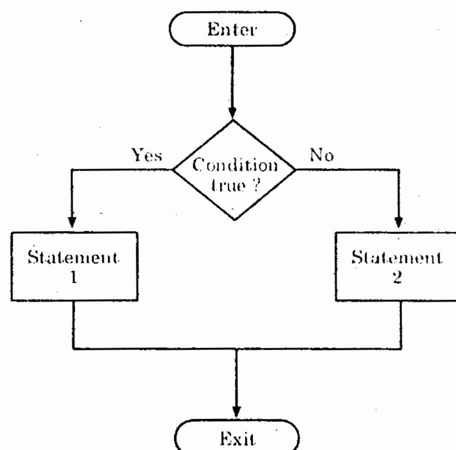
Giả mã :

```
IF [ condition ]
        THEN [ statement 1 ]
        ELSE [ statement 2 ]
```

Statement : phát biểu

Condition : điều kiện

Lưu đồ : ( hình 8.8 )




Hình 8.8 : Lưu đồ của cấu trúc IF / THEN / ELSE

**Enter : nhập**

**Condition true ? : điều kiện đúng ?**

**Statement : phát biểu**

**Exit : thoát**

***Thí dụ 8.5*** : Kiểm tra ký tự

*Hãy viết một chuỗi lệnh để nhập và kiểm tra một ký tự từ port nối tiếp. Nếu ký tự là mã ASCII hiển thị được ( nghĩa là trong tầm từ 20H đến 7EH ), cho hiển thị ký tự, ngược lại cho hiển thị ký tự ( . ).*

***Lời giải***

Giả mã :

```
[ input character ]
IF  [ character == graphic ]
        THEN  [ echo character ]
        ELSE  [ echo ' . ' ]
```

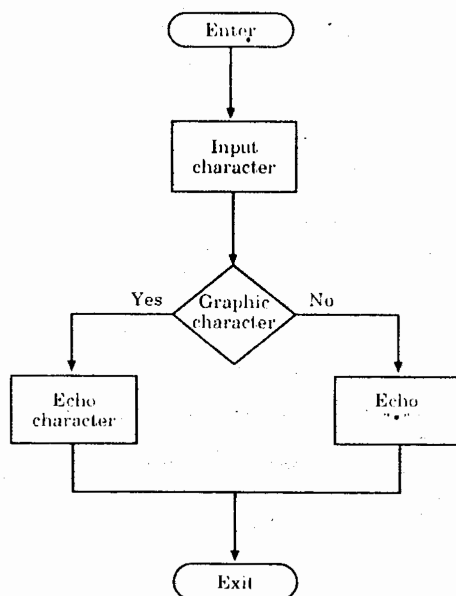
Input character : nhập ký tự

Character == graphic : ký tự thuộc loại ký tự đồ họa

Echo character : hiển thị ký tự

Echo ' . ' : hiển thị dấu ' . '

Lưu đồ : ( hình 8.9 )




Hình 8.9 : Lưu đồ của thí dụ 8.5

Enter : nhập

Input character : nhập ký tự

Graphic character ? : ký tự thuộc loại ký tự đồ họa ?

Echo character : hiển thị ký tự

Echo ' . ' : hiển thị dấu ' . '

Exit : thoát

Chương trình cho 8051 :

```
                (được cấu trúc chặt : 14 byte)
ENTER:          ACALL       INCH
                ACALL       ISGRPH
                JNC         STMENT2
STMENT1:        ACALL       OUTCHR
                JMP         EXIT
STMENT2:        MOV         A, # '.'
                ACALL       OUTCHR
EXIT:           ( continue )
                (được cấu trúc lỏng : 10 byte)
                ACALL       INCH
                ACALL       ISGRPH
                JC          SKIP
                MOV         A, # '.'
SKIP            ACALL       OUTCHR
                ( continue )
```

Sửa lại cấu trúc để lặp lại không hạn định :

```
WHILE [ 1 ] DO BEGIN
                [ input character ]
        IF [ character == graphic ]
                THEN [ echo character ]
                ELSE [ echo ' . ' ]
END
```

Input character : nhập ký tự

Character == graphic : ký tự thuộc loại ký tự đồ họa

Echo character : hiển thị ký tự

Echo ' . ' : hiển thị dấu ' . '

### 8.3.3.2 Phát biểu CASE

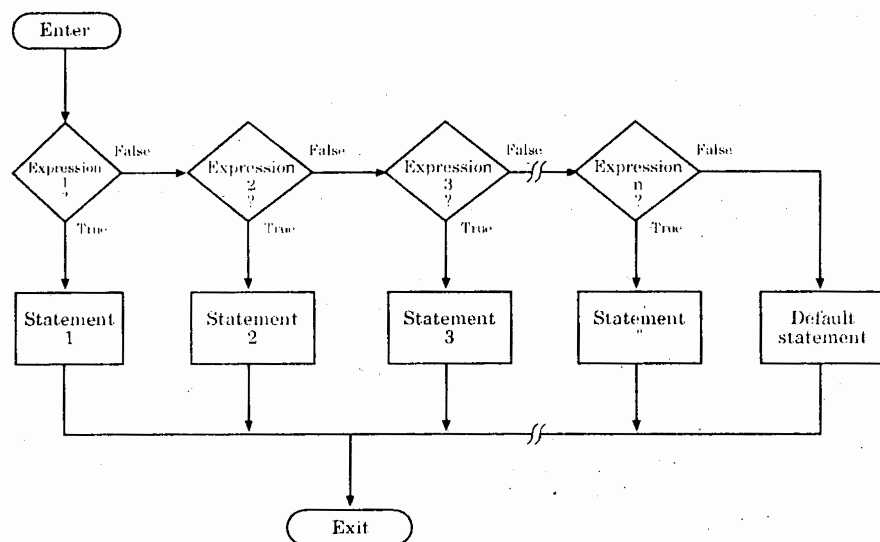
Phát biểu CASE là biến thể dễ sử dụng của phát biểu IF / THEN / ELSE. Phát biểu CASE được sử dụng khi một trong nhiều phát biểu phải được lựa chọn. Việc lựa chọn dựa vào việc xác định một giá trị.

Giả mã :

```
CASE [ expression ] OF
        0 : [ statement 0 ]
        1 : [ statement 1 ]
        2 : [ statement 2 ]
        .
        .
        n : [ statement n ]
        ( default statement )
END_CASE
```

Lưu đồ :

( xem hình 8.10 )




Hình 8.10 : Lưu đồ của cấu trúc CASE

Enter : nhập

Expression : biểu thức

Statement : phát biểu

Default statement : phát biểu mặc định

Exit : thoát

***Thí dụ 8.6*** : Đáp ứng của người sử dụng

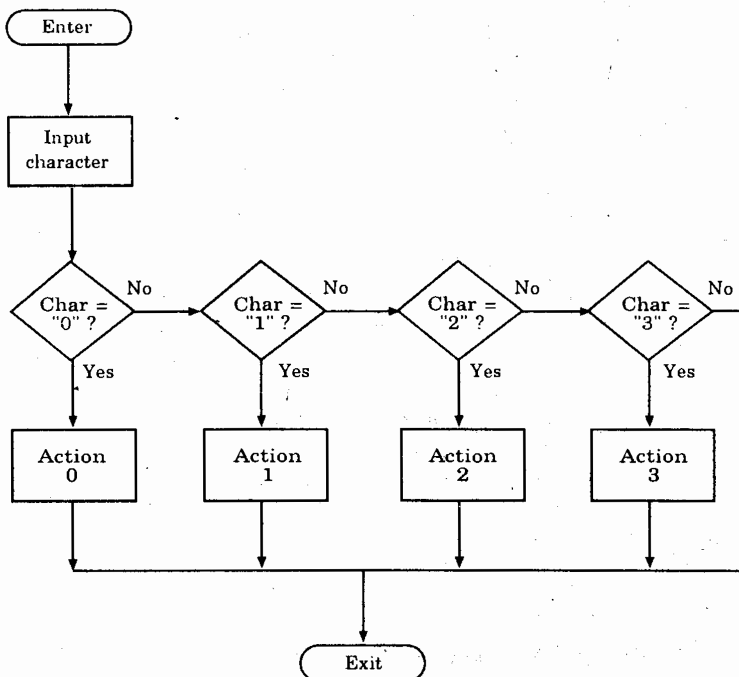
*Một chương trình được điều khiển bởi một thực đơn yêu cầu người sử dụng trả lời là 0, 1 2, hoặc 3 để lựa chọn 1 trong 4 thao tác. Hãy viết một chuỗi lệnh để nhập một ký tự từ bàn phím và nhảy đến ACT0, ACT1, ACT2 hoặc ACT3 phụ thuộc vào ký tự do người sử dụng nhập vào. Bỏ qua việc kiểm tra lỗi.*

***Lời giải***

Giả mã :

```
[ input character ]
CASE [ character ] OF
        '0' : [ statement 0 ]
        '1' : [ statement 1 ]
        '2' : [ statement 2 ]
        '3' : [ statement 3 ]
END_CASE
```

Lưu đồ : ( hình 8.11 )




**Hình 8.11 : Lưu đồ của thí dụ 8.6**

Enter : nhập

Input character : nhập ký tự

Char = ‘ 0 ‘ ? : ký tự là 0 ?

Action 0 : thao tác 0

Exit : thoát

Chương trình cho 8051 :

( được cấu trúc chặt )

```
                CALL   INCH
                CJNE   A, # '0', SKIP1
    ACT0:       .
                .
                JMP    EXIT
    SKIP1:      CJNE   A, # '1', SKIP2
    ACT1:       .
                .
                JMP    EXIT
    SKIP2:      CJNE   A, # '2', SKIP3
    ACT2:       .
                .
                JMP    EXIT
    SKIP3:      CJNE   A, # '3', EXIT
    ACT3:       .
                .
    EXIT:       ( continue )
```

( được cấu trúc lỏng )

```
                CALL   INCH
                ANL    A, # 3
                RL     A
                MOV    DPTR, #TABLE
                JMP    @A + DPTR
    TABLE :     AJMP   ACT0
                AJMP   ACT1
```

```
            AJMP  ACT2
ACT3:       .
            .
            JMP   EXIT
ACT0:       .
            .
            JMP   EXIT
ACT1:       .
            .
            JMP   EXIT
ACT2:       .
            .
EXIT:       ( continue )
```

#### 8.3.3.3 Phát biểu GOTO

Phát biểu GOTO có thể không dùng đến một khi đã sử dụng các cấu trúc vừa mới trình bày. Tuy nhiên phát biểu GOTO lại cho ta một phương pháp dễ dàng để kết thúc một cấu trúc khi có lỗi xuất hiện và dĩ nhiên ta cần phải hết sức cẩn thận. Khi một chương trình được viết bằng hợp ngữ, phát biểu GOTO luôn luôn trở thành một lệnh nhảy không điều kiện. Một vấn đề sẽ phát sinh, thí dụ một thủ tục được gọi bằng lệnh gọi thủ tục CALL nhưng ta lại thoát khỏi thủ tục bằng cách sử dụng một lệnh nhảy thay vì là lệnh trở về từ thủ tục, địa chỉ quay về vẫn còn tồn tại trong *stack* và cuối cùng có khả năng xảy ra tràn *stack*.

## 8.4 CÚ PHÁP CỦA GIẢ MÃ

Vì giả mã tương tự như ngôn ngữ cấp cao và đáng được định nghĩa một cách chính thức sao cho một chương trình viết bằng giả mã bởi một lập trình viên có thể được biến đổi thành chương trình hợp ngữ bởi một lập trình viên khác.

Ta cần biết rằng giả mã không phải luôn luôn là phương pháp tốt nhất cho việc thiết kế chương trình. Trong khi giả mã cho ta lợi ích về việc cấu trúc dễ dàng trên một bộ xử lý từ ( word processor ), giả mã cũng tạo ra các bất lợi đối với các ngôn ngữ lập trình khác : chương trình giả mã được viết từng dòng một, do vậy các thao tác song song hiển nhiên không tức thời. Bằng cách dùng lưu đồ các thao tác song song có thể được đặt kề nhau, do vậy cải tiến được cách nhận thức ( xem hình 8.10 ).

Trước khi giới thiệu cú pháp chính thức của giả mã, các điều sau đây được nêu ra để tăng cường khả năng giải quyết các bài tập lập trình sử dụng giả mã.

- sử dụng ngôn ngữ mô tả cho các phát biểu.
- tránh các phụ thuộc vào máy trong các phát biểu.
- đóng các điều kiện và các phát biểu trong dấu ngoặc vuông [ ]
- bắt đầu mọi chương trình con với tên của chúng được theo sau bởi một tập các dấu ngoặc tròn ( ). Các thông số được truyền đến các chương trình con được đưa vào ( bằng tên hay giá trị ) trong các dấu ngoặc tròn này.
- kết thúc mọi thủ tục bằng RETURN được theo sau bởi các dấu ngoặc tròn. Các giá trị trả về được đưa vào trong các dấu ngoặc tròn này.

Các thí dụ :

```
INCHAR ( )
        [ statement ]
        . . . . .
RETURN ( char )
OUTCHR ( char )
        [ statement ]
        . . . . .
RETURN ( )
STRLEN ( pointer )
        [ statement ]
        . . . . .
RETURN ( length )
```

- sử dụng chữ thường ngoại trừ các từ khóa và tên của các thủ tục.
- các phát biểu được thụt vào so với các điểm nhập và thoát của cấu trúc. Khi cấu trúc LOOP hoặc CHOOSE được bắt đầu, các phát biểu trong cấu trúc xuất hiện ở vị trí thụt vào tiếp theo.
- sử dụng ký tự @ cho các địa chỉ gián tiếp.

Sau đây là cú pháp được đề nghị cho giả mã :

*Các từ khóa :*

```
BEGIN      END
REPEAT     UNTIL
WHILE      DO
IF         THEN      ELSE
CASE       OF
RETURN
```

*Các toán tử số học :*

```
+     cộng
-     trừ
*     nhân
/     chia
%     modulus
```

*Các toán tử quan hệ :*

```
==    đúng nếu bằng
!=    đúng nếu không bằng
<     đúng nếu nhỏ hơn
<=    đúng nếu nhỏ hơn hay bằng
>     đúng nếu lớn hơn
>=    đúng nếu lớn hơn hay bằng
&&    đúng nếu cả hai cùng đúng
||    đúng nếu một trong hai đúng
```

*Các toán tử logic :*

```
&     AND
|     OR
^     XOR
~     NOT
>>    dịch phải
<<    dịch trái
```

*Các toán tử gán :*

= thiết lập bằng với

op = thao tác gán với op là một trong các toán tử :

+ – * / % << >> & ^ |

nghĩa là : j + = 4 tương đương với j = j + 4

*Thao tác ưu tiên :*

( )

*Địa chỉ gián tiếp :*

@

*Ưu tiên của các toán tử :*

( )

~ @

* / %

+ –

<< >>

< <= > >=

== !=

&

^

|

&&

= += –= *= v.v...

<u>Lưu ý 1</u> : Ta không nên nhầm lẫn các toán tử quan hệ với các toán tử logic. Một cách tổng quát, các toán tử logic được sử dụng trong các phát biểu gán như là :

[ lower_nibble = byte & 0FH ]

trong khi đó các toán tử quan hệ được dùng trong các biểu thức điều kiện như :

IF ( char != ‘ Q ‘ && char != 0DH ) THEN . . . .

<u>Lưu ý 2</u> : Ta không nên nhầm lẫn toán tử quan hệ “ == “ với toán tử gán “ = “. Thí dụ biểu thức :

```
j == 9
```

hoặc có giá trị đúng hoặc có giá trị sai phụ thuộc vào giá trị của j so với 9. Trong khi đó phát biểu :

```
j = 9
```

gán giá trị 9 cho biến j.

*Các cấu trúc :*

Phát biểu :

```
( do something )
```

Khối phát biểu :

```
BEGIN
        [ statement ]
        [ statement ]
        . . . . .
END
```

WHILE / DO :

```
WHILE  [ condition ]  DO
        [ statement ]
```

REPEAT / UNTIL :

```
REPEAT
        [ statement ]
UNTIL  [ condition ]
```

IF / THEN / ELSE :

```
IF [ condition ]
        THEN  [ statement 1 ]
        ( ELSE [ statement 2 ] )
```

CASE / OF

```
CASE  [ expression ]  OF
        1 : [ statement 1 ]
        2:  [ statement 2 ]
```

.

.

n : [ statement n ]

[ default statement ]

Condition : điều kiện

Statement : phát biểu

Default : mặc định

## 8.5 LẬP TRÌNH HỢP NGỮ

Vấn đề quan trọng trong lập trình hợp ngữ là chọn cách lập trình rõ ràng và phù hợp. Điều này đặc biệt quan trọng khi ta làm việc trong một nhóm và từng người trong nhóm phải có thể đọc và hiểu từng chương trình của những người khác.

Các lời giải bằng hợp ngữ cho các bài tập cho đến lúc này là sơ sài một cách cố ý. Với các công việc lập trình lớn hơn, một phương pháp quan trọng hơn được cần đến. Các vấn đề sau được đưa ra nhằm giúp ta cải thiện cách lập trình hợp ngữ.

### 8.5.1 Nhãn

Các nhãn cần có tên sao cho mô tả được mục đích sử dụng, thí dụ khi ta rẽ nhánh trở về để lặp lại việc thực hiện một thao tác, ta nên sử dụng nhãn có tên như là " LOOP ", " BACK ", " MORE " v.v... Khi ta bỏ qua một vài lệnh trong chương trình, nên sử dụng nhãn có tên như là " SKIP " hoặc " AHEAD ". Khi ta lặp lại việc kiểm tra một bit trạng thái, nên sử dụng nhãn có tên như là " WAIT " hoặc " AGAIN ".

Việc lựa chọn tên cho các nhãn bị hạn chế một ít khi sử dụng một trình dịch hợp ngữ tuyệt đối hoặc thường trú trong bộ nhớ. Các trình dịch hợp ngữ này xử lý toàn bộ chương trình như là một đơn vị nên giới hạn việc đặt tên cho các nhãn có cùng mục đích. Một vài kỹ thuật loại bỏ được vấn đề này. Tên của các nhãn chung có chung mục đích có thể được đánh số thứ tự như là SKIP1, SKIP2, SKIP3, v.v... hoặc trong các thủ tục, tên của các nhãn có thể là tên của thủ tục được tiếp theo bằng một con số như là SEND, SEND2, SEND3 v.v... Ở đây tính rõ ràng dĩ nhiên bị mất vì các nhãn SEND2 và SEND3 không thể phản ánh được việc bỏ qua hay lặp vòng.

Nhiều trình dịch hợp ngữ tinh vi hơn như ASM51 cho phép mỗi một thủ tục ( hoặc một nhóm các thủ tục ) hiện hữu trong các tập tin riêng rẽ, được hợp dịch độc lập với chương trình chính. Chương trình chính cũng được hợp dịch và rồi kết hợp với các thủ tục bằng cách sử dụng

trình liên kết và định vị, trình này giải quyết các tham chiếu ngoài giữa các tập tin. Loại trình dịch hợp ngữ này thường được gọi là trình dịch hợp ngữ tái định vị cho phép các nhãn có cùng mục đích có tên giống nhau vì chúng xuất hiện trong các tập tin khác nhau.

### 8.5.2 Chú thích

Việc sử dụng các chú thích không được quá nhấn mạnh, đặc biệt là trong lập trình hợp ngữ, vốn dĩ đã trừu tượng. Tất cả các dòng lệnh, ngoại trừ những dòng có các hoạt động hiển nhiên nên có thêm trường chú thích.

Các lệnh nhảy có điều kiện được chú thích một cách có hiệu quả bằng cách sử dụng câu hỏi tương tự với câu hỏi trong lưu đồ đối với cùng một thao tác. Các câu trả lời " phải ", đúng " ( " yes " ) và " không ", " sai " ( " no " ) cho câu hỏi nên xuất hiện trong các chú thích ở các dòng biểu diễn các hoạt động nhảy và không nhảy. Thí dụ trong chương trình con INLINE dưới đây, ta hãy chú ý cách mà các chú thích được sử dụng để kiểm tra mã ASCII của <CR>.

```
;**********************************************
; INLINE    INPUT LINE OF CHARACTERS
;           nhập dòng ký tự
;           LINE MUST END WITH <CR>
;           dòng phải được kết thúc bằng < CR >
;           MAXIMUM LENGTH 31 CHARACTERS INCLUDE <CR>
;           chiều dài tối đa của dòng là 31 ký tự bao gồm cả <CR>
;
; ENTER:    NO CONDITIONS
;           không có các điều kiện
; EXIT:     ASCII CODES IN INTERNAL DATA RAM
;           các mã ASCII chứa trong RAM dữ liệu nội
;           0 STORED AT END OF LINE
;           0 được chứa ở cuối dòng
; USES      INCHAR, OUTCHR
; sử dụng các chương trình con INCHAR và OUTCHR
;**********************************************
;
```

```
INLINE     PUSH   00H              ; lưu R0 vào stack
           PUSH   07H              ; lưu R7 vào stack

           PUSH   ACC              ; lưu thanh chứa vào stack
           MOV    R0, #60H         ; thiết lập vùng đệm ở 60H
           MOV    R7, #31          ; chiều dài cực đại của dòng
STMENT:    ACALL  INCHAR           ; nhập một ký tự
           ACALL  OUTCHR           ; hiển thị lên VDT
           MOV    @R0, A           ; lưu vào vùng đệm
           INC    R0               ; tăng con trỏ vùng đệm
           DEC    R7               ; giảm bộ đếm chiều dài
           CJNE   A, #0DH, SKIP    ; có phải ký tự là <CR> ?
           SJMP   EXIT             ; phải : thoát
SKIP:      CJNE   R7, #0, STMENT; không : nhập ký tự khác
EXIT:      MOV    @R0, #0
           POP    ACC              ; khôi phục các thanh ghi
           POP    07H              ; từ stack
           POP    00H
           RET
```

### 8.5.3 Khối chú thích

Các dòng chú thích cần thiết nên được đặt ở nơi bắt đầu mỗi một chương trình con. Vì các chương trình con thực hiện các công việc được xác định rõ ràng và được cần đến trong suốt chương trình, chúng nên có mục đích tổng quát và được dẫn chứng rõ ràng. Mỗi một chương trình con nên có một khối chú thích ( các dòng chú thích ) đứng trước và nội dung khối này cần nêu rõ :

- tên của chương trình con
- thao tác được thực hiện
- các điều kiện để vào chương trình con
- các điều kiện ra khỏi chương trình con
- tên của các chương trình con khác được dùng đến ( nếu có )
- tên các thanh ghi bị ảnh hưởng bởi chương trình ( nếu có )

Chương trình con INLINE ở trên là một thí dụ điển hình cho một chương trình con được chú thích tốt.

### 8.5.4 Lưu các thanh ghi vào *stack*

Với các ứng dụng có kích thước lớn và phức tạp, các chương trình con mới thường được viết dựa trên các chương trình con đã có. Do vậy một chương trình con có thể gọi một chương trình con khác rồi đến lượt chương trình con bị gọi này lại gọi một chương trình con khác nữa, v.v... Điều này được gọi là các chương trình con được lồng vào nhau ( nested subroutines ). Không có gì đáng lo ngại trong việc lồng các chương trình con nếu như *stack* có đủ cửa sổ để lưu giữ các địa chỉ quay về. Đây không là vấn đề vì hiếm khi việc lồng vào nhau vượt quá một vài mức.

Vấn đề quan trọng là việc sử dụng các thanh ghi trong các chương trình con. Khi mà hệ thống thứ bậc của các chương trình con phát triển ta sẽ ngày càng khó theo dõi các thanh ghi bị ảnh hưởng bởi các chương trình con. Một thói quen lập trình đáng tin cậy là cất các thanh ghi vào *stack* nếu chúng bị thay đổi bởi chương trình con và sau đó phục hồi lại chúng ở cuối chương trình con. Lưu ý là chương trình con INLINE trình bày ở trên cất và phục hồi R0, R7 và thanh chứa. Khi INLINE trở về chương trình gọi, các thanh ghi này chứa các giá trị giống như khi INLINE được gọi.

### 8.5.5 Gán

Việc định nghĩa các hằng số bằng phát biểu gán làm cho chương trình dễ đọc và dễ bảo trì hơn. Các gán xuất hiện ở đầu chương trình để định nghĩa các hằng số hoặc các địa chỉ của các thanh ghi bên trong các IC giao tiếp như là thanh ghi trạng thái STATUS, thanh ghi điều khiển CONTROL.

Các hằng số được dùng trong suốt chương trình bằng cách thay thế các giá trị bằng các ký hiệu đã được gán. Khi chương trình được hợp dịch, các giá trị tương ứng được thay thế cho các ký hiệu. Việc sử dụng phép gán làm cho chương trình dễ bảo trì hơn cũng như dễ đọc hơn. Nếu một hằng số cần được thay đổi, ta chỉ phải thay đổi một dòng – dòng mà ở đó ký hiệu được gán. Khi chương trình được hợp dịch lại, giá trị mới tự động được thay thế ở những nơi mà ký hiệu được sử dụng.

### 8.5.6 Các chương trình con

Khi các chương trình trở nên lớn, ta phải chia nhỏ các thao tác lớn và phức tạp thành các thao tác nhỏ và đơn giản. Các thao tác nhỏ và đơn giản này được lập trình thành các chương trình con. Các chương

trình con đều có thứ bậc, trong đó các chương trình con đơn giản có thể được sử dụng bởi các chương trình con phức tạp hơn và v.v...

Lưu đồ :

Một lưu đồ tham chiếu một chương trình con bằng cách dùng hộp " xử lý tiền định nghĩa " ( xem hình 8.1 ). Việc sử dụng ký hiệu này chỉ ra rằng một lưu đồ ở một nơi khác mô tả các chi tiết của thao tác.

Các chương trình con được cấu trúc từ giả mã là các phần đầy đủ của chương trình, bắt đầu bằng tên của chương trình con và các dấu ngoặc tròn. Bên trong các dấu ngoặc tròn là các tên và các giá trị của các tham số được truyền tới chương trình con ( nếu có ). Mỗi một chương trình con được kết thúc bằng từ khóa RETURN, được tiếp theo bởi các dấu ngoặc tròn chứa tên và giá trị của các tham số được trả về bởi chương trình con ( nếu có ).

Có lẽ thí dụ về thứ bậc chương trình con đơn giản nhất là các chương trình con xuất chuỗi ( OUTSTR ) và xuất ký tự ( OUTCHR ). Chương trình con OUTSTR ( chương trình mức cao ) gọi chương trình con OUTCHR ( chương trình mức thấp ).

Lưu đồ, giả mã và lời giải hợp ngữ cho 8051 được trình bày dưới đây.

Giả mã :

```
OUTCHR ( char )
        [ put odd parity in bit 7 ]
        REPEAT          [ test transmit buffer ]
                UNTIL   [ buffer empty ]
        [ clear transmit buffer empty flag ]
        [ move char to transmit buffer ]
        [ clear parity bit ]
RETURN ( )
OUTSTR ( pointer )
        WHILE [ ( char = @pointer ) != 0 ]  BEGIN
                OUTCHR ( char )
                [ increment poiter ]
        END
RETURN ( )
```

Put odd parity in bit 7 : đặt bit kiểm tra lẻ vào bit 7

Test transmit buffer : kiểm tra bộ đệm phát

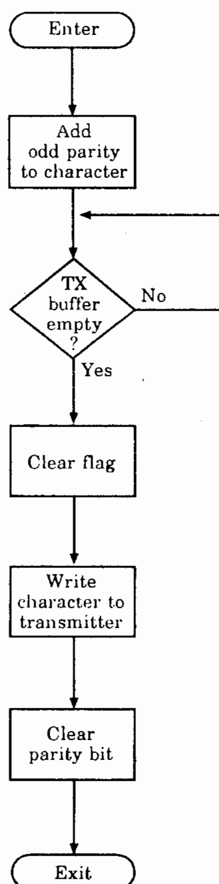
Buffer empty : bộ đệm rỗng

Clear transmit buffer empty flag : xóa cờ báo bộ đệm phát rỗng

Move char to transmit buffer : di chuyển ký tự đến bộ đệm phát

Clear parity bit : xóa bit kiểm tra chẵn lẻ

Increment poiter : tăng nội dung con trỏ




Hình 8.12 : Lưu đồ của chương trình con OUTCHR

Enter : nhập

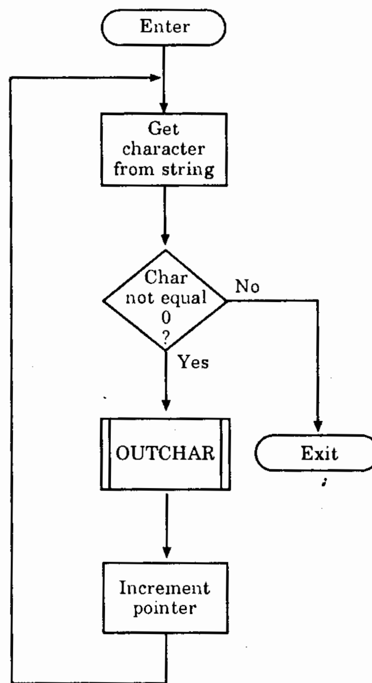
Add odd parity to character : thêm bit kiểm tra lẻ vào ký tự

TX buffer empty ? : Bộ đệm TX rỗng ? Clear flag : xóa cờ

Write character to transmitter : ghi ký tự vào bộ phát

Clear bit parity : xóa bit kiểm tra chẵn lẻ

Exit : thoát




Hình 8.13 : Lưu đồ của chương trình con OUTSTR

Enter : nhập

Get character from string : lấy ký tự từ chuỗi

Char not equal 0 ? : char là số 0 ?

Increment poiter : tăng nội dung con trỏ

Exit : thoát

Chương trình cho 8051 :

```
OUTCHR:   MOV   C, P          ; đặt bit kiểm tra vào cờ C
          CPL   C             ; đổi thành kiểm tra lẻ
          MOV   ACC.7, C      ; cộng vào ký tự
AGAIN:    JNB   TI, AGAIN     ; TX rỗng ?
          CLR   TI            ; phải : xóa cờ và
          MOV   SBUF, A       ; gởi ký tự
          CLR   ACC.7         ; bỏ bit kiểm tra và
          RET                 ; quay về chương trình gọi
```

```
OUTSTR:     MOV     A, @DPTR        ; lấy một ký tự
            JZ      EXIT            ; nếu 0, kết thúc
            CALL    OUTCHR          ; ngược lại gởi ký tự đi
            INC     DPTR            ; tăng con trỏ
            SJMP    OUTSTR          ; và lấy ký tự kế
EXIT:       RET
```

### 8.5.7 Tổ chức chương trình

Mặc dù các chương trình thường được viết từng phần ( nghĩa là các chương trình con được viết riêng rẽ với chương trình chính ), mọi chương trình nên thích hợp với tổ chức sau cùng. Một cách tổng quát, các phần của chương trình được sắp xếp theo thứ tự sau :

- các phép gán
- các lệnh khởi động
- thân chính của chương trình
- các chương trình con
- các định nghĩa hằng dữ liệu ( DB và DW )
- các vị trí dữ liệu trong RAM được định nghĩa bằng chỉ dẫn DS

5 thành phần đầu thuộc *segment* mã còn thành phần sau cùng thuộc *segment* dữ liệu. *Segment* mã và *segment* dữ liệu được phân biệt với nhau theo thông lệ do chương trình thường được đặt trong ROM hoặc EPROM trong khi dữ liệu thường được đặt trong RAM. Lưu ý là các chuỗi và hằng dữ liệu được định nghĩa bằng các chỉ dẫn DB và DW là một phần của *segment* mã ( không phải của *segment* dữ liệu ) vì các dữ liệu này là các hằng số không thay đổi và do vậy là một phần của chương trình.

# 9

# THIẾT KẾ VÀ GIAO TIẾP

## 9.1 MỞ ĐẦU

Chương này giới thiệu các đặc trưng phần cứng cũng như phần mềm của 8051 thông qua các thí dụ về thiết kế và giao tiếp. Trước tiên ta khảo sát một máy tính 8-bit đơn *board* ( single-board ) mang tên SBC-51 thích hợp cho việc nghiên cứu 8051 và phát triển các sản phẩm dựa trên 8051. SBC-51 sử dụng một chương trình *monitor* cung cấp các lệnh điều khiển ( command ) cơ bản cho hoạt động của hệ thống và tác động qua lại với người sử dụng. Chương trình *monitor* này ( MON51 ) được mô tả chi tiết ở phụ lục G.

Các thí dụ về giao tiếp được nâng cao bằng cách so sánh với các thí dụ đã được trình bày trong các chương trước. Mỗi một thí dụ bao gồm sơ đồ phần cứng, phát biểu về mục tiêu thiết kế, liệt kê chương trình thực hiện mục tiêu thiết kế và mô tả tổng quát hoạt động của phần cứng và phần mềm. Các liệt kê chương trình được chú giải một cách bao quát và các chi tiết cụ thể được quan tâm đến.

## 9.2 SBC-51

Một vài công ty cung cấp các máy tính đơn *board* dựa trên 8051 tương tự như máy tính được mô tả trong mục này. Điều đáng ngạc nhiên là thiết kế cơ bản của một máy tính đơn *board* dùng 8051 không thay đổi nhiều giữa các sản phẩm khác nhau do các công ty khác nhau cung cấp. Do có nhiều đặc trưng được đặt bên trong chip 8051, việc thiết kế một máy tính đơn *board* dùng 8051 không phức tạp, chỉ có các kết nối cơ bản đến bộ nhớ ngoài và giao tiếp với máy tính là cần đến.

Ta cũng cần đến chương trình *monitor* được chứa trong ROM. Các yêu cầu của hệ cơ bản này như là kiểm tra và thay đổi các vị trí nhớ hoặc nạp các chương trình ứng dụng từ máy tính xuống được cần đến. SBC-51 được mô tả ở đây cùng hoạt động với một chương trình *monitor* đơn giản nhằm cung cấp các chức năng cơ bản vừa nêu.

Hình 9.1 là sơ đồ mạch của SBC-51. Toàn bộ thiết kế chỉ bao gồm 10 vi mạch ( IC ), tuy nhiên thiết kế này cũng đủ khả năng và linh hoạt để hỗ trợ cho sự phát triển các sản phẩm phức tạp dựa trên 8051. Hoạt động của 8051 chủ yếu dựa vào chương trình *monitor* thường trú trong EPROM và việc truyền thông với một thiết bị đầu cuối hiển thị *video* VDT ( video display terminal ) kết nối với 8051.

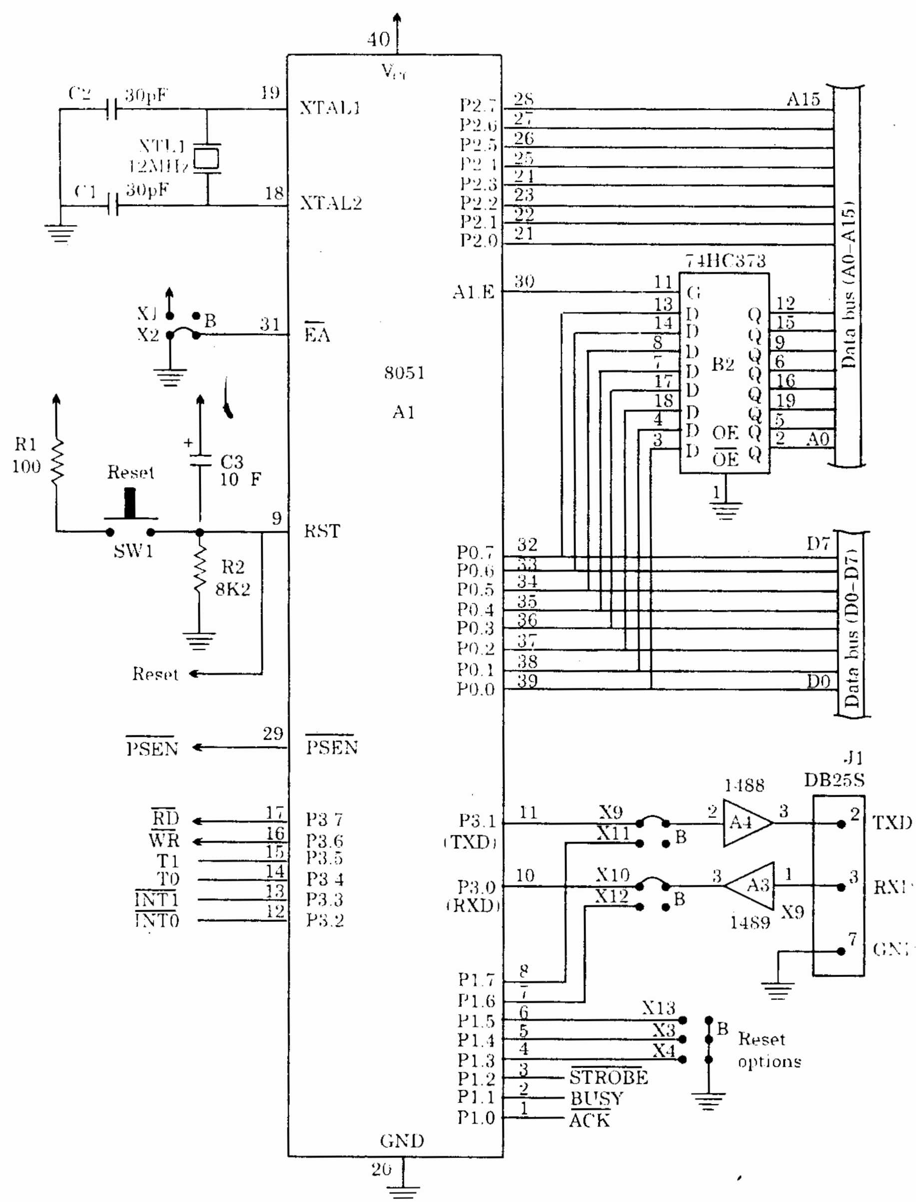
SBC-51 bao gồm thêm , ngoài các đặc trưng chuẩn của 80C31, 16 KB EPROM ngoài, 8.25 KB RAM ngoài, bộ định thời 14-bit và 22 đường xuất nhập. Cấu hình được trình bày ở hình 9.1 bao gồm các linh kiện và thành phần sau :

- 10 vi mạch
- 15 tụ điện
- 2 điện trở
- 1 thạch anh
- 1 chuyển mạch nút nhấn
- 3 đầu nối
- 13 *jumper* định cấu hình

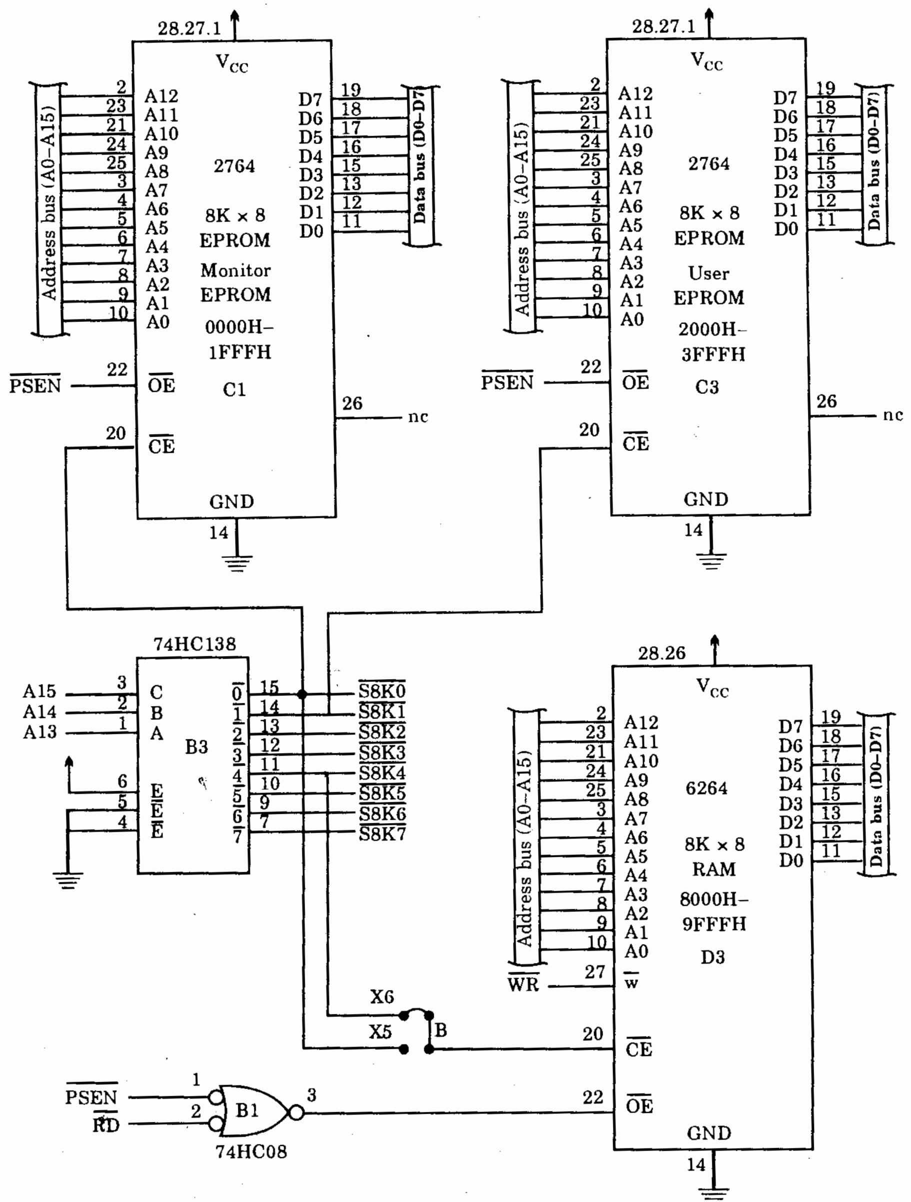
Vì phải sử dụng bộ nhớ ngoài, *port* 0 và *port* 2 không còn được dùng để xuất nhập. Do các *port* 1 và *port* 3 chỉ sử dụng một phần cho các đặc trưng cụ thể, một số đường của *port* 1 và *port* 3 có thể được sử dụng cho mục đích xuất nhập, tùy thuộc vào cấu hình.

Nguồn xung *clock* cho 80C31 là một thạch anh 12 MHz được kết nối theo cách thông thường với mạch dao động bên trong 80C31 ( xem hình 2.2 ). Chân RST ( reset ) được điều khiển bởi một mạch R-C để *reset* hệ thống tự động khi cấp nguồn, đồng thời ta có thể *reset* hệ thống bằng tay nhờ vào một chuyển mạch nút nhấn. *Port* 0 là *port* đa hợp hai nhiệm vụ, vừa là bus dữ liệu ( D0 đến D7 ) vừa là byte thấp của bus địa chỉ ( A0 đến A7 ) như đã đề cập đến ở mục 2.6 của chương 2. Mạch chốt 8-bit 74HC373 được điều khiển chốt bởi tín hiệu ALE sẽ duy trì byte thấp của bus địa chỉ trong trong suốt chu kỳ bộ nhớ. Vì 80C31 không có ROM trong chip, chương trình thực thi phải chứa trong ROM ngoài nên chân $\overline{EA}$ phải nối với 0 V nhờ *jumper* cấu hình X2.

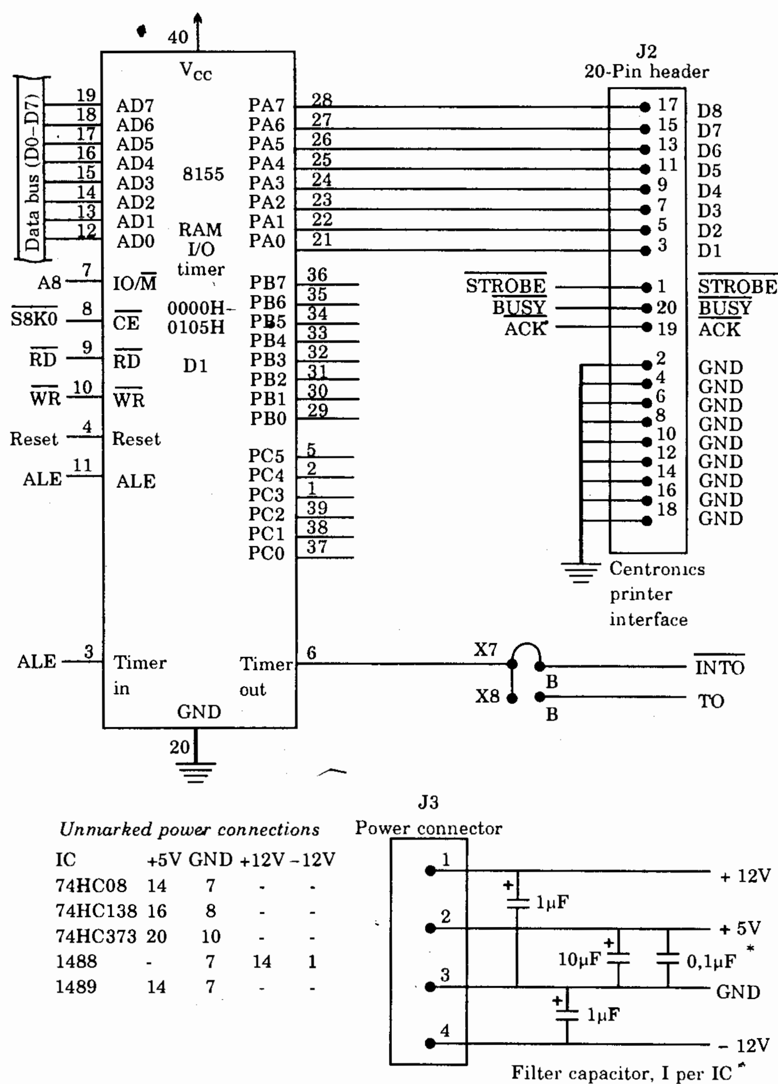
Việc kết nối với máy tính hoặc VDT sử dụng giao tiếp nối tiếp RS232C. Đầu nối DB25 được nối dây trở thành một DTE ( data terminal equipment : thiết bị đầu cuối dữ liệu ) với dữ liệu phát trên chân TxD ( chân 2 ), dữ liệu thu trên chân RxD ( chân 3 ) và chân 7 nối đất ( 0 V ).




Hình 9.1 : Máy tính đơn board sử dụng 8051




Hình 9.1 : ( tiếp theo )




**Hình 9.1 : ( tiếp theo )**

Mạch kích đường truyền ( line driver ) RS232C 1488 nối với chân TxD và mạch thu đường truyền ( line receiver ) RS232C 1489 nối với chân RxD. Việc kết nối mặc định với 80C31 thông qua các *jumper* X9 và X10 là : P3.1 làm nhiệm vụ của chân TxD và P3.0 làm nhiệm vụ của chân RxD. Một cách tùy chọn, thông qua các *jumper* X11 và X12, các chức năng TxD và RxD có thể được cung cấp thông qua phần mềm bằng cách sử dụng các chân P1.7 và P1.6.

Các đường 3, 4 và 5 của *port* 1 được đọc bởi chương trình *monitor* trong lúc *reset* để yêu cầu các đặc trưng đặc biệt. Tuy nhiên sau khi *reset*, các đường này trở thành các đường xuất nhập. Nếu giao tiếp máy in được sử dụng, các đường 0, 1 và 2 của *port* 1 được dùng làm các tín hiệu bắt tay. Nếu giao tiếp máy in không sử dụng, các đường này làm nhiệm vụ xuất nhập.

Vi mạch 74HC138 giải mã 3 bit cao của bus địa chỉ ( A13 đến A15 ) và tạo ra 8 đường có công dụng lựa chọn, mỗi đường lựa chọn một khối bộ nhớ 8 KB. Các đường này được gọi là S8K0 ( chọn khối 8 K thứ 0 ) đến S8K7 ( chọn khối 8 K thứ 7 ). Bốn IC được lựa chọn bởi các đường này : hai EPROM 2764, một RAM 6264 và một bộ RAM/xuất nhập/định thời ( RAM/IO/TIMER ) 8155.

Hai EPROM 8KB 2764 được trình bày trong hình 9.1. EPROM đầu tiên ( gọi là MONITOR EPROM ) được chọn bởi đường S8K0 và thường trú trong không gian bộ nhớ chương trình ngoài từ địa chỉ 0000H đến 1FFFH. Vì SBC-51 sẽ bắt đầu thực thi chương trình từ địa chỉ 0000H ngay sau khi *reset* hệ thống, chương trình *monitor* phải thường trú trong IC này. EPROM 8KB 2764 thứ hai ( gọi là USER EPROM ) được chọn bởi đường S8K1 và có địa chỉ từ 2000H đến 3FFFH. IC này được dự định dành cho các ứng dụng của người sử dụng và không cần thiết cho hoạt động của hệ cơ bản. Lưu ý là cả hai EPROM được chọn chỉ nếu $\overline{CE}$ ( chip enable : cho phép chip, chân 20 ) ở trạng thái tích cực ( hoặc thấp ) và $\overline{OE}$ cũng ở trạng thái tích cực ( hoặc thấp ). $\overline{OE}$ được điều khiển bởi đường $\overline{PSEN}$ của 80C31, nghĩa là không gian bộ nhớ chương trình ngoài được chọn.

RAM 8KB 6264 được chọn bởi đường S8K4 ( nếu *jumper* X6 được cài đặt như trong hình vẽ ) và có địa chỉ từ 8000H đến 9FFFH. RAM được chọn sẽ chiếm cả hai không gian bộ nhớ ngoài : dữ liệu và chương trình bằng cách dùng phương pháp đã mô tả trước đây ở mục 2.6.4 của chương 2. Điều này cho phép các chương trình của người sử dụng được nạp ( hoặc ghi ) tới RAM như là bộ nhớ dữ liệu và rồi được thực thi như là bộ nhớ chương trình.

RAM/IO/TIMER 8155 là IC giao tiếp ngoại vi được thêm vào để chứng minh khả năng mở rộng của SBC-51. Ta cũng dễ dàng cộng thêm các IC giao tiếp ngoại vi khác theo cách tương tự. 8155 được chọn bởi đường S8K0, như vậy được đặt ở đáy của bộ nhớ. Không có xung đột xảy ra với MONITOR EPROM ( cũng được đặt ở đáy của bộ nhớ, nhưng trong không gian bộ nhớ chương trình ngoài ) bởi vì 8155 được điều khiển đọc / ghi bằng các tín hiệu $\overline{RD}$ và $\overline{WR}$.

8155 chứa các đặc trưng sau :

- 256 byte RAM
- 22 đường xuất nhập
- bộ định thời 14-bit

Đường địa chỉ A8 nối với đường IO/$\overline{M}$ của 8155 ( chân 7 ), đường này chọn RAM khi ở mức thấp và chọn các đường xuất nhập hoặc bộ định thời khi ở mức cao. Các đường xuất nhập và bộ định thời được truy xuất từ 6 địa chỉ, do vậy tầm địa chỉ tổng cộng của 8155 từ 0000H đến 0105H ( 256 + 6 địa chỉ ). Dưới đây là tóm tắt của tầm địa chỉ :

| **Địa chỉ** | **Mục đích** |
|---|---|
| 0000H | Địa chỉ đầu tiên của RAM |
| . . . . | Các địa chỉ kế tiếp |
| 00FFH | Địa chỉ sau cùng của RAM |
| 0100H | Thanh ghi lệnh / định thời |
| 0101H | Port A |
| 0102H | Port B |
| 0103H | Port C |
| 0104H | 8 bit thấp cho số đếm định thời |
| 0105H | 6 bit cao cho số đếm định thời & 2 bit cho chế độ định thời |

Mặc dù ta nên tham khảo thêm tài liệu kỹ thuật của nhà sản xuất để biết các chi tiết hoạt động của 8155, việc cấu hình cho các *port* xuất nhập lại rất dễ dàng. Do được mặc định tất cả các đường của *port* là đường nhập sau khi *reset* hệ thống, ta không cần khởi động để đọc các thiết bị nhập được nối với 8155. Thí dụ để đọc *port* A vào thanh chứa, chuỗi lệnh sau đây được dùng :

```
MOV     DPTR, #0101H        ; DPTR trỏ đến port A của 8155
MOVX    A, @DPTR            ; đọc port A vào ACC
```

Để lập trình cho *port* A và *port* B xuất, các bit 1 trước tiên phải được ghi vào thanh ghi lệnh điều khiển ( command register ) ở các bit 0 và bit 1. Thí dụ để cấu hình cho *port* B xuất còn các *port* A và *port* C vẫn làm nhiệm vụ nhập, chuỗi lệnh sau được dùng :

```
MOV   DPTR, 0100H      ; thanh ghi lệnh điều khiển của
                       ; 8155
MOV   A, #00000010B    ; port B xuất
MOVX  @DPTR, A         ; khởi động 8155
```

*Port* C được cấu hình làm *port* xuất bằng cách ghi 1 vào thanh ghi lệnh điều khiển ở các bit 2 và bit 3. Cả 3 *port* được cấu hình làm các *port* xuất như sau :

```
MOV   DPTR, 0100H      ; thanh ghi lệnh điều khiển của
                       ; 8155
MOV   A, #00001111B    ; các port đều xuất
MOVX  @DPTR, A         ; khởi động 8155
```

*Port* A của 8155 như trên hình vẽ được kết nối với một *header* 20-pin có tên là " Centronics printer interface " ( giao tiếp máy in Centronics ). Giao tiếp này chỉ nhằm minh họa các mục đích xuất nhập. MON51 bao gồm một chương trình con PCHAR ( print character : in ký tự ) hướng ngõ ra đến VDT và một máy in song song nếu CONTROL-Z được giữ và gõ từ bàn phím ( xem phụ lục G ). Dĩ nhiên *port* A có thể được sử dụng cho các mục đích khác nếu cần.

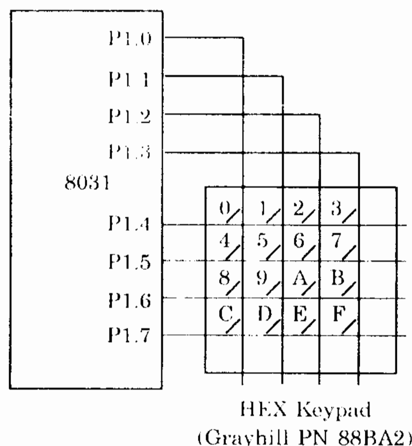
Các kết nối với nguồn cấp điện cũng được vẽ trên hình 9.1. Các tụ lọc đặc biệt quan trọng đối với điện áp + 5 V nhằm tránh các gai nhiễu tạo ra bởi các hiệu ứng cảm khi các linh kiện số chuyển trạng thái. Ta hãy đặt một tụ điện giải 10 μF ở nơi nối nguồn cấp điện với *board* mạch và tụ gốm 0.01 μF bên cạnh đế cắm của mỗi một IC, tụ được nối giữa chân 5 V và chân nối đất.

Đến đây ta kết thúc phần mô tả SBC-51. Các mục tiếp theo bao gồm các thí dụ giao tiếp với các thiết bị ngoại vi, chúng được phát triển để kết nối với SBC-51 ( hoặc một máy tính đơn *board* tương tự dùng 8051 ).

## 9.3 GIAO TIẾP VỚI BÀN PHÍM SỐ HEX

Giao tiếp với bàn phím thường được cần đến đối với các thiết kế dựa trên bộ vi điều khiển. Nhập từ bàn phím và xuất ra LED là sự lựa chọn kinh tế để giao tiếp với người sử dụng và thường thích hợp với các ứng dụng phức tạp. Các thí dụ bao gồm việc giao tiếp người sử dụng với lò vi ba hoặc máy đổi tiền tự động.

Hình 9.2 trình bày cách giao tiếp giữa *port* 1 và bàn phím số hex. Bàn phím có 16 phím được sắp xếp thành 4 hàng và 4 cột. Các đường hàng được nối với các bit từ 4 đến 7 còn các đường cột được nối với các bit từ 0 đến 3 của *port* 1.




Hình 9.2 : Giao tiếp với bàn phím số hex

## Mục tiêu thiết kế

*Viết một chương trình đọc liên tiếp các ký tự số hex từ bàn phím và cho hiển thị dưới dạng mã ASCII trên VDT*

Thoạt nhìn thí dụ này có vẽ khá đơn giản. Chương trình được chia thành các bước sau :

1. Nhập ký tự số hex từ bàn phím

2. Biến đổi mã số hex thành mã ASCII

3. Gởi ký tự ASCII đến VDT

4. Quay về bước 1

Việc lập trình theo đúng các bước trên và chương trình được trình bày ở hình 9.3 ( dòng 16 – 19 ). Dĩ nhiên các công việc được thực hiện trong các chương trình con. Lưu ý là các bước 2 và 3 ở trên được hiện thực bằng cách gọi các chương trình con trong MON51. Ta có thể trích các chương trình con nêu trên từ MON51 và đặt chúng vào trong chương trình ở hình 9.3, nhưng điều này thật phí phạm. Thay vào đó, các điểm nhập của MON51 cho các chương trình con này được xác định ở gần trên đỉnh của chương trình ( dòng 12 – 13 ) bằng cách sử dụng các ký hiệu HTOA và OUTCHR. Sau đó các chương trình con được gọi từ vòng lặp của chương trình chính MAIN theo cách thông thường. Các điểm nhập cho các chương trình con của MON51 có thể được tìm thấy trong bảng

ký hiệu được tạo ra bởi RL51 khi MON-51 được liên kết và định vị. Các điểm nhập của HTOA và OUTCHR được tìm thấy trong phụ lục G.

Thách thức thực sự của thí dụ này là viết các chương trình con IN_HEX và GET_KEY. GET_KEY thực hiện việc quét các hàng và cột của bàn phím để xác định có phải một phím được ấn. Nếu không có phím nào được ấn, GET_JEY trả về C = 0 còn nếu có một phím được ấn, GET_KEY trả về C = 1, mã số hex của phím được đưa vào thanh chứa ở các bit từ 0 đến 3.

IN_HEX thực hiện việc chống xung nẩy khi ấn và nhả phím bằng phần mềm. Vì bàn phím là một chuỗi các chuyển mạch cơ khí, việc tiếp xúc khi ấn và nhả phím bao gồm cả xung nẩy xảy ra nhanh và ngắn. Việc chống xung nẩy được thực hiện bằng cách lặp lại việc gọi GET_HEX cho đến khi 50 lần gọi liên tiếp đều trả về C = 1. Lần gọi nào đó trả về C = 0 đều được hiểu là nhiễu ( nghĩa là có xung nẩy ) và bộ đếm được *reset*. Sau khi phát hiện một phím được ấn hợp lệ, IN_HEX chờ 50 lần gọi liên tiếp GET_HEX trả về C = 0 để đảm bảo rằng phím đã hoàn toàn được nhả trước khi gọi GET_HEX lần nữa cho phím được ấn tiếp theo.

Chương trình ở hình 9.3 làm việc được nhưng không thật sự tốt. Do các ngắt không được sử dụng, tiện ích của chương trình trong một ứng dụng lớn hơn bị hạn chế. Do vậy một cải tiến hợp lý là thiết kế lại phần mềm và sử dụng ngắt. Giao tiếp được điều khiển ngắt sẽ được minh họa trong thí dụ kế.

```
$DEBUG
$NOPAGING
$NOSYMBOLS
; FILE : KEYPAD.SRC
;*********************************************
;                   KEYPAD INTERFACE EXAMPLE
;
; Chương trình này đọc các ký tự số hex từ một bàn phím nối với port 1
; và
; cho hiển thị các phím được ấn lên VDT
;*********************************************
HTOA        EQU   033CH        ; các chương trình con của
OUTCHR      EQU   01DEH        ; MON51
```

Hình 9.3 : Chương trình giao tiếp bàn phím

```
                ORG     8000H
MAIN:           CALL    IN_HEX          ; đọc mã từ bàn phím
                CALL    HTOA            ; biến đổi thành mã ASCII
                CALL    OUTCHR          ; hiển thị lên VDT
                SJMP    MAIN            ; lặp lại

;*****************************************************************
; IN_HEX : nhập mã số hex từ bàn phím có chống nẩy khi ấn và nhả
; phím
; ( lặp lại 50 lần cho ấn phím và cho nhả phím )
;*****************************************************************
IN_HEX:         MOV     R3, #50         ; số đếm
BACK:           CALL    GET_KEY         ; phím được ấn ?
                JNC     IN_HEX          ; không : kiểm tra lại
                DJNZ    R3, BACK        ; có: lặp lại 50 lần
                PUSH    ACC             ; lưu mã số hex
BACK2:          MOV     R3, #50         ; chờ phím nhả
BACK3:          CALL    GET_KEY         ; phím được ấn ?
                JC      BACK2           ; có : kiểm tra lại
                DJNZ    R3, BACK3       ; không : lặp lại 50 lần
                POP     ACC             ; khôi phục số hex và
                RET                     ; quay về

;*****************************************************************
; GET_KEY :     - đọc trạng thái bàn phím
;               - trả về C = 0 nếu không có phím ấn
;               - trả về C = 1 và mã số hex trong ACC
;               nếu có phím ấn
;*****************************************************************
GET_KEY:        MOV     A, #0FEH        ; bắt đầu với cột 0
                MOV     R6, #4          ; sử dụng R6 làm bộ đếm
TEST:           MOV     P1, A           ; tích cực cột
                MOV     R7, A           ; lưu ACC
                MOV     A, P1           ; đọc trở lại port 1
```

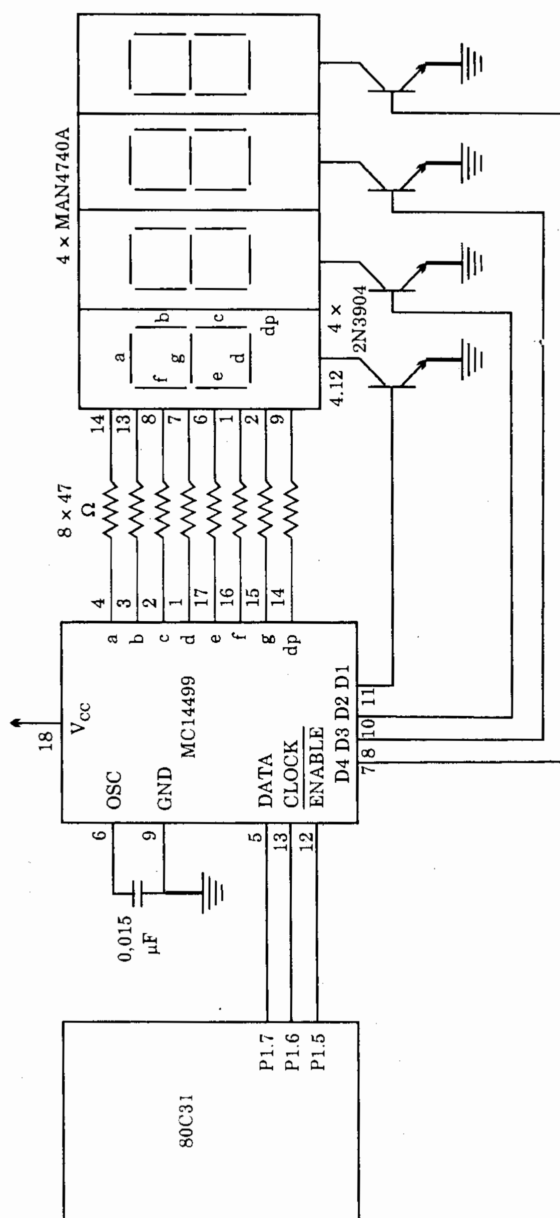
Hình 9.3 ( tiếp theo )

```
            ANL     A, #0F0H            ; cách ly các hàng
            CJNE    A, #0F0H, KEY_HIT   ; hàng tích cực ?
            MOV     A, R7               ; không : di chuyển tới
            RL      A                   ;         cột kế
            DJNZ    R6, TEST            ;
            CLR     C                   ; không có phím được ấn
            SJMP    EXIT                ; quay về với C = 0
KEY_HIT:    MOV     R7, A               ; lưu trong R7
            MOV     A, #4               ; chuẩn bị tính
            CLR     C                   ; trọng số của cột
            SUBB    A, R6               ; 4 – R6 = trọng số
            MOV     R6, A               ; lưu trong R6
            MOV     A, R7               ; phục hồi mã quét
            SWAP    A                   ; đặt trong 4 bit thấp
            MOV     R5, #4              ; dùng R5 làm bộ đếm
AGAIN:      RRC     A                   ; quay cho đến khi = 0
            JNC     DONE                ; thực thi xong khi C = 0
            INC     R6                  ; cộng 4 cho đến khi
            INC     R6                  ; tìm thấy hàng tích cực
            INC     R6
            INC     R6
            DJNZ    R5, AGAIN
DONE:       SETB    C                   ; C = 1 ( có phím được ấn )
            MOV     A, R6               ; mã trong A
EXIT:       RET
            END
```

Hình 9.3 ( tiếp theo )

## 9.4 GIAO TIẾP VỚI CÁC ĐÈN 7-ĐOẠN

Giao tiếp với một đèn 7-đoạn đã được trình bày trong một bài tập ở cuối chương 3 ( xem hình 3.5 ), nhưng do mạch giao tiếp sử dụng 7 đường của *port* 1 nên mạch này biểu thị một sự phân phối xấu các tài nguyên trên *chip*. Trong mục này ta minh họa một mạch giao tiếp với bốn đèn 7-đoạn chỉ sử dụng ba trong số các đường xuất nhập của 8051. Hiển nhiên đây là một thiết kế được cải tiến, đặc biệt là khi có nhiều đèn 7-đoạn được kết nối. Trung tâm của thiết kế này là vi mạch giải mã và kích đèn 7-đoạn MC14499 của Motorola, vi mạch này chứa bên trong nhiều mạch cần thiết cho việc kích 4 đèn 7-đoạn.




Hình 9.4 : Giao tiếp với MC14499 và 4 đèn 7-đoạn

Các thành phần được thêm vào chỉ là tụ định thời 0.015 μF, 7 điện trở giới hạn dòng 47 Ω và 4 transistor 2N3904. Hình 9.4 trình bày cách kết nối giữa 80C31, vi mạch MC14499 và 4 đèn 7-đoạn.

**Mục tiêu thiết kế**

*Giả sử các digit BCD được chứa trong RAM nội tại các vị trí 70H và 71H. Sao chép các digit BCD đến các đèn 7-đoạn 10 lần trong 1 sec sử dụng ngắt.*

Chương trình thực hiện mục tiêu trên được trình bày trong hình 9.5.

Chương trình này minh họa một số khái niệm đã được thảo luận trước đây. Các chi tiết mức thấp để phát dữ liệu đến MC14499 được tìm thấy trong các chương trình con UPDATE và OUT8. Ở mức cao hơn, thí dụ này minh họa việc thiết kế các ứng dụng được điều khiển ngắt có tầm quan trọng đối với hoạt động ở mức nền và mức ngắt ( không giống như các thí dụ ở chương 6 chỉ hoạt động ở mức ngắt ). Các ngắt cho thí dụ này cùng tồn tại với MON51, bản thân chương trình này không sử dụng các ngắt. Chương trình *monitor* thực thi ở mức nền trong khi chương trình ở hình 9.5 thực thi ở mức ngắt.

Khi chương trình được bắt đầu ( bằng cách đưa vào MON51 lệnh GO8000; xem phụ lục G ), các điều kiện được khởi động để cập nhật ngắt ban đầu cần thiết của các đèn 7-đoạn, sau đó điều khiển nhanh chóng quay trở lại chương trình *monitor*. Các lệnh điều khiển của chương trình *monitor* được thực thi theo cách thông thường trong khi chờ đợi các ngắt sẽ xuất hiện ở mức ngắt. Thí dụ nếu lệnh SET của chương trình *monitor* được sử dụng để thay đổi các vị trí 70H và 71H của RAM nội, sự thay đổi này được thể hiện ngay tức khắc ( trong vòng 0.1 sec ) trên các đèn 7 đoạn.

Hãy lưu ý đến cấu trúc tổng thể của chương trình. Các phần sau đây xuất hiện theo thứ tự :

- các điều khiển của trình dịch hợp ngữ ( các dòng 1 – 3 )
- khối chú thích ( các dòng 4 – 30 )
- định nghĩa các ký hiệu ( các dòng 31 – 38 )
- các khai báo định nghĩa vùng nhớ ( các dòng 40 – 42 )
- bảng nhảy cho chương trình và các điểm nhập của các ngắt ( các dòng 44 – 51 )
- phần chính ( MAIN; các dòng 56 – 69 )
- trình phục vụ ngắt ngoài ( EXT0ISR; các dòng 74 – 77 )

- chương trình con cập nhật các đèn 7-đoạn ( UPDATE; các dòng 89 – 97 )
- chương trình con xuất byte ( OUT8; các dòng 103 – 113 )
- đoạn chương trình quản lý các ngắt không được hiện thực ( các dòng 118 – 123 )

Chương trình được viết để được thực thi ở địa chỉ 8000H trong IC RAM 6264 của SBC-51. Vì các vector ngắt đặt tại các vị trí ở đáy của bộ nhớ, chương trình *monitor* bao gồm một bảng nhảy định hướng lại các ngắt đến các địa chỉ bắt đầu ở địa chỉ 8000H ( xem phụ lục G ). Điểm nhập của chương trình là 8000H nhưng một lệnh nhảy dài LJMP ( dòng 45 ) chuyển điều khiển đến nhãn MAIN. Tất cả các lệnh khởi động được bao gồm trong các dòng từ 56 đến 68. Phần MAIN kết thúc bằng việc nhảy trở về chương trình *monitor*.

```
1   $DEBUG
2   $NOPAGING
3   $NOSYMBOLS
4   ; FILE : MC14499.SRC
5   ;*****************************************
6   ;                  MC14499 INTERFACE EXAMPLE
7   ;
8   ; Chương trình này cập nhật việc hiển thị 4-digit 10 lần trong 1 sec sử
9   ; dụng ngắt.
10  ; Các digit là các đèn 7-đoạn được kích bởi vi mạch giải mã / kích
11  ; kết nối với P1.5 ( ENABLE ), P1.6 ( CLOCK ) và P1.7 (DATA  IN)
12  ; Các ngắt được tạo ra bởi đường TIMER OUT của 8155 nối với INT0.
13  ; TIMER OUT dao động ở tần số 500 Hz và tạo ra một ngắt khi có một
14  ; chuyển đổi trạng thái 1 → 0.
15  ; Một bộ đếm ngắt được sử dụng để cập nhật việc hiển thị sau mỗi 50
16  ; ngắt,
17  ; như vậy tần số cập nhật là 10 Hz.
18  ;
19  ; Thí dụ này minh họa các khái niệm mức nền và mức ngắt đối với các
20  ; hệ được điều khiển ngắt.
```

Hình 9.5 : Phần mềm giao tiếp với MC14499

```
;
; THOÁT :     MC14499 được cập nhật
; DÙNG :      P1.5, P1.6, P1.7
;             Tất cả vị trí nhớ và các thanh ghi không ảnh hưởng
;*****************************************
UPDATE:       PUSH    ACC           ; cất thanh chứa vào stack
              CLR     ENABLE        ; chuẩn bị MC14499
              MOV     A, DIGITS     ; lấy 2 digit đầu tiên
              ACALL   OUT8          ; gởi ra 2 digit LED
              MOV     A, DIGITS+1   ; lấy byte hai
              ACALL   OUT8          ; gởi ra 2 digit LED kế
              SETB    ENABLE        ; không cho phép MC14499
              POP     ACC           ; khôi phục ACC từ stack
              RET

;*****************************************
;  SEND 8 BITS IN ACCUMULATOR TO MC14499 ( MSB FIRST )
;*****************************************
              USING   0             ; giả sử dãy 0 được phép
OUT8:         PUSH    AR7           ; cất R7 vào stack
              MOV     R7, #8        ; sử dụng R7 làm bộ đếm bit
AGAIN:        RLC     A             ; đặt bit vào cờ C
              MOV     DIN, C        ; gởi bit đến MC14499
              CLR     CLOK          ; xung có 3 µs ở mức thấp
              NOP                   ; NOP cần để kéo dài xung
              NOP                   ; ( độ rộng xung tối thiểu
              SETB    CLOCK         ; là 2 µsec )
              DJNE    R7, AGAIN     ; lập lại đến khi 8 bit được gởi
              POP     AR7           ; khôi phục R7 từ stack
              RET
```

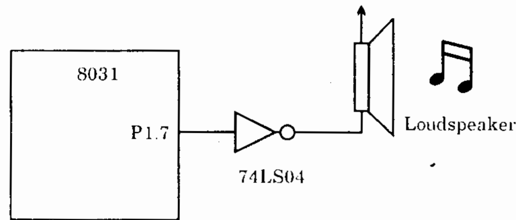
Hình 9.5 : ( tiếp theo )

```
;****************************************
; UNUSED INTERRUPTS ( ERROR ; RETURN TO MONITOR PROG
;****************************************
T0ISR:
EX1ISR:
T1ISR:
SPISR
T2ISR:      CLR     EA          ; cấm các ngắtvà
            LJMP    MON51       ; trở về MON51
            END
```

Hình 9.5 : ( tiếp theo )

## 9.5 GIAO TIẾP VỚI LOA

Hình 9.6 trình bày giao tiếp giữa 8031 và loa. Các loa nhỏ, chẳng hạn các loa trong các máy tính cá nhân hoặc đồ chơi trẻ em, có thể được kích từ một cổng logic như được trình bày trên hình. Một đầu cuộn dây của loa ghép với + 5 V, đầu còn lại ghép với ngõ ra của cổng đảo 74LS04. Cổng đảo được cần đến do cổng này có khả năng cấp và hút dòng cao hơn các chân *port* của 8031.




Hình 9.6 : Giao tiếp với loa

### Mục tiêu thiết kế

*Viết một chương trình được điều khiển ngắt phát liên tục gam nhạc LA trưởng.*

Các giai điệu nhạc dễ dàng được tạo ra từ 8051 bằng cách sử dụng mạch giao tiếp loa đơn giản. Chúng ta bắt đầu với một ít nhạc lý. Tần số cho mỗi nốt trong gam nhạc LA trưởng được cho trong khối chú thích ở đầu danh sách liệt kê phần mềm ( hình 9.7 các dòng từ 14 đến 21 ). Tần số đầu tiên là 440 Hz là tần số của nốt LA. Tần số của tất cả các nốt khác có thể được xác định bằng cách nhân tần số này cho $2^{n/12}$, trong đó n là số các bước ( hoặc các ½ cung ) tính từ nốt LA đến nốt

được tính. Thí dụ dễ nhất là nốt A', một bát độ hoặc 12 bước trên A, tần số của nốt này là 440 x $2^{12/12}$ bằng 880 Hz. Đây là nốt sau cùng trong gam nhạc của chúng ta. ( xem hình 9.7 dòng 21 ). Bằng cách tham chiếu đến nốt ở đáy ( hay gốc ) trong một gam nhạc trưởng bất kỳ, tỉ lệ cho các bước là 2, 4, 5, 7, 9, 11 và 12. Lấy thí dụ nốt E trong hình 9.7 ( dòng 18 ) cách 7 bước trên gốc, vậy thì tần số của nốt này là 440 x $2^{7/12}$ = 659.26 Hz.

Để tạo ra một gam nhạc, hai khoảng thời gian định thời được cần đến : khoảng thời gian từ một nốt đến nốt kế và khoảng thời gian bit của *port* chuyển đổi trạng thái để kích loa. Hai khoảng thời gian định thời này khác nhau. Thí dụ để chơi một giai điệu nhạc ở tốc độ 4 nốt trong 1 sec, một ngắt ( do tràn bộ định thời ) được cần đến sau mỗi một 250 msec. Để tạo tần số cho nốt đầu tiên trong gam nhạc, một ngắt được cần đến sau mỗi một 1.136 msec ( xem hình 9.7 dòng 14 )

Phần mềm ở hình 9.7 khởi động cả hai bộ định thời ở chế độ định thời 16-bit ( dòng 4 ) và sử dụng ngắt do bộ định thời 0 cho việc thay đổi nốt và ngắt do bộ định thời 1 cho tần số của các nốt. Các giá trị nạp lại cho các tần số của nốt được đọc từ bảng tìm kiếm ( các dòng từ 90 đến 104 ).

```
1   $DEBUG
2   $NOPAGING
3   $NOSYMBOLS
4   ; FILE : SCALE.SRC
5   ;****************************************
6   ;           LOUDSPEAKER INTERFACE EXAMPLE
7   ;
8   ; Đây là chương trình phát một gam nhạc LA trưởng sử dụng một loa
9   ; được kích bởi một cổng đảo ghép với P1.7
10  ;****************************************
11  ;
12  ;     Nốt   Tần số ( Hz )   Chu kỳ ( µsec )   Chu kỳ / 2 ( µsec )
13  ;     ---   ---------       -----------       -------------
14  ;     A     440.00          2273              1136
15  ;     B     493.88          2025              1012
16  ;     C#    554.37          1804              902
```

Hình 9.7 : Phần mềm giao tiếp với loa

```
;          D        587.33          1703            851
;          E        659.26          1517            758
;          F#       739.99          1351            676
;          G#       830.61          1204            602
;          A'       880.00          1136            568
;*****************************************************
MONITOR    CODE     00BCH           ; điểm nhập của MON51
COUNT      EQU      - 50000         ; 0.05 sec cho một ngắt
REPEAT     EQU      5               ; 5 x 0.05 = 0.25 sec / nốt

;*****************************************************
; Chú ý : X3 không được cài đặt trên SBC-51, do vậy các ngắt
; trỏ đến bảng nhảy theo sau
; bắt đầu ở 8000H
;*****************************************************
           ORG      8000H           ; các điểm nhập cho :
           LJMP     MAIN            ; chương trình chính
           LJMP     EXT0ISR         ; ngắt ngoài 0
           LJMP     T0ISR           ; ngắt do bộ định thời 0
           LJMP     EXT1ISR         ; ngắt ngoài 1
           LJMP     T1ISR           ; ngắt do bộ định thời 1
           LJMP     SPISR           ; ngắt do port nối tiếp
           LJMP     T2ISR           ; ngắt do bộ định thời 2

;*****************************************************
; MAIN PROGRAM BEGINS
;*****************************************************
MAIN:      MOV      TMOD, #11H      ; hai bộ định thời ở chế độ 1
           MOV      R7, #0          ; sử dụng R7 làm bộ đếm nốt
           MOV      R6, #REPEAT     ; sử dụng R6 làm bộ đếm ngắt
           MOV      IE, #8AH        ; các ngắt do bộ định thời 0, 1
           SETB     TF1             ; buộc ngắt do bộ định thời 1
```

Hình 9.7 : ( tiếp theo )

```
                SETB    TF0             ; buộc ngắt do bộ định thời 0
                SJMP    $

;**************************************************
;   TIMER 0 INTERRUPT SERVICE ROUTINE ( EVERY 0.05 sec )
;**************************************************
T0ISR:          CLR     TR0             ; ngừng bộ định thời
                MOV     TH0, #HIGH (COUNT)  ; nạp
                MOV     TL0, #LOW (COUNT)   ; lại
                DJNZ    R6, EXIT        ; không phải ngắt 5, thoát
                MOV     R6, #REPEAT     ; là ngắt 5, reset
                INC     R7              ; tăng nốt
                CJNE    R7, #LENGTH, EXIT   ; nốt sau cùng ?
                MOV     R7, #0          ; phải : reset, A = 440 Hz
EXIT:           SETB    TR0             ; không : bắt đầu bộ định thời
                RETI

;**************************************************
; TIMER 1 INTERRUPT SERVICE ROUTINE ( PITCH OF NOTES )
;
; Lưu ý : Các tần số ngưng do độ dài của ISR này.
; Việc nạp lại bộ định thời cần được hiệu chỉnh
;
;**************************************************
T1ISR:          CPL     P1.7            ;
                CLR     TR1             ; ngừng bộ định thời
                MOV     A, R7           ; đọc bộ đếm nốt
                RL      A               ; nhân ( 2 byte / nốt )
                CALL    GETBYTE         ; lấy byte cao của số đếm
                MOV     TH1, A          ; đặt vào TH1
                MOV     A, R7           ; đọc bộ đếm lần nữa
                RL      A
```

Hình 9.7 : ( tiếp theo )

```
            INC     A
            CALL    GETBYTE         ; lấy byte thấp của số đếm
            MOV     TL1, A          ; đặt vào TL1
            SETB    TR1             ; bắt đầu bộ định thời
            RETI

;****************************************
; GET A BYTE FROM LOOK-UP OF NOTES IN A MAJOR SCALE
;****************************************
GETBYTE:    INC     A               ; chương trình con tìm kiếm
            MOVC    A, @A+PC        ; bảng
            RET
TABLE:      DW      - 1136          ; A
            DW      - 1136          ; A lần nữa ( ½ nốt )
            DW      - 1012          ; B ( ¼ nốt  v.v... )
            DW      - 902           ; C#
            DW      - 851           ; D
            DW      - 758           ; E
            DW      - 676           ; F#
            DW      - 602           ; G#
            DW      - 568           ; A'
            DW      - 568           ; A' ( 4 lần, cả nốt )
            DW      - 568
            DW      - 568
LENGTH      EQU     ($ - TABLE)/2   ; LENGTH = # của nốt

;****************************************
; UNUSED INTERRUPT – BACK TO MONITOR PROGRAM (ERROR)
;****************************************
EXT0ISR:
EXT1ISR:
SPISR:
```

Hình 9.7 : ( tiếp theo )

```
113 T2ISR:     CLR     EA          ; cấm các ngắt và
114            LJMP    MONITOR     ; trở về MON51
115            END
```

Hình 9.7 : ( tiếp theo )

## 9.6 GIAO TIẾP VỚI RAM KHÔNG MẤT NỘI DUNG

RAM không mất nội dung NV-RAM ( non-volatile RAM ) là bộ nhớ bán dẫn duy trì được nội dung khi ta không cung cấp điện cho RAM. NVRAM kết hợp cả hai : các phần tử của RAM tĩnh và các phần tử ROM lập trình được và xóa được ( EEROM ). Mỗi một bit của RAM tĩnh được phủ bởi một bit của EEROM. Dữ liệu có thể truyền qua lại giữa hai bit nhớ và như vậy giữa hai bộ nhớ.

NVRAM chiếm một vị trí quan trọng trong các ứng dụng của bộ vi xử lý và bộ vi điều khiển. NVRAM được sử dụng để lưu các dữ liệu và các tham số được cài đặt, các dữ liệu và thông số này thỉnh thoảng được thay đổi bởi người sử dụng nhưng phải được duy trì khi không được cung cấp điện.

Thí dụ nhiều thiết kế của VDT không sử dụng các chuyển mạch DIP mà sử dụng NVRAM để lưu trữ các thông tin được cài đặt như là tốc độ baud, cho phép hoặc không cho phép bit chẵn lẻ, kiểm tra chẵn hay lẻ v.v... Mỗi khi VDT được cấp điện, các tham số này được gọi từ NVRAM và hệ thống được khởi động tương ứng một cách thích hợp. Khi một tham số được thay đổi bởi người sử dụng ( thông qua bàn phím ), giá trị mới được lưu trong NVRAM.

Các *modem* có đặc trưng tự động quay số thường lưu các số điện thoại trong bộ nhớ nội. Các số điện thoại này thường được lưu trong NVRAM. Mười số điện thoại với 7 digit cho mỗi số có thể được lưu trong 35 byte ( bằng cách mã hóa mỗi một digit ở dạng số BCD )

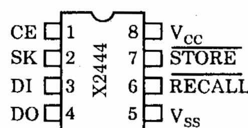
NVRAM được sử dụng trong thí dụ giao tiếp này là X2444 của Xicor, một công ty chuyên sản xuất NVRAM và EEROM. X2444 chứa 256 bit RAM tĩnh được phủ bởi 256 bit EEROM. Các dữ liệu được truyền qua lại giữa hai bit nhớ hoặc bằng các lệnh được gởi đi từ bộ xử lý trên mạch giao tiếp nối tiếp hoặc bằng cách sử dụng hai ngõ vào STORE và RECALL. Các dữ liệu không bị mất được lưu trong EEROM trong khi dữ liệu độc lập được truy xuất và cập nhật trong RAM. Các đặc trưng của X2444 như sau :

- Lý tưởng khi sử dụng với các máy vi tính đơn chip
- định thời tĩnh

- giao tiếp I/O tối thiểu
- tương thích với *port* nối tiếp ( COPS™, 8051 )
- dễ dàng giao tiếp với các *port* của các bộ vi điều khiển
- các mạch hỗ trợ tối thiểu

- Đặc trưng không mất thông tin được điều khiển bởi phần cứng và phần mềm
  - bảo vệ bộ nhớ tối đa
- Tương thích TTL
- Tổ chức 16 x 16
- Công suất tiêu tán nhiệt thấp
  - dòng tích cực : điển hình 15 mA
  - dòng lưu trữ : điển hình 8 mA
  - dòng chờ : điển hình 6 mA
  - dòng nghỉ : điển hình 5 mA

Sơ đồ các chân ra được trình bày ở hình 9.8



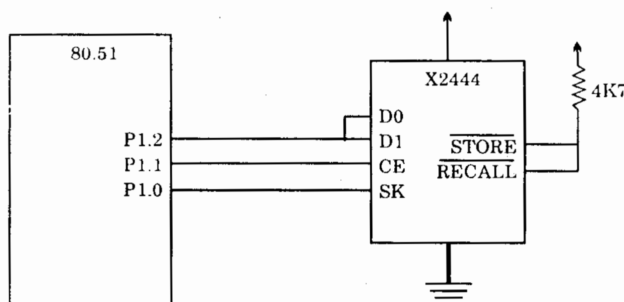

| | |
|---|---|
| CE | Chip Enable |
| SK | Serial Clock |
| DI | Serial Data In |
| DO | Serial Data Out |
| $\overline{RECALL}$ | Recall |
| $\overline{STORE}$ | Store |
| $V_{CC}$ | + 5V |
| $V_{SS}$ | Ground |

Hình 9.8 : Các thông số kỹ thuật của X2444

Trong thí dụ giao tiếp này, các đường STORE và RECALL không được sử dụng. Các chế độ khác của thao tác được đưa vào bằng cách gởi đến X2444 các lệnh ở dạng nối tiếp thông qua các chân *port* của 8051.

Mạch giao tiếp với 8051 được trình bày ở hình 9.9.

Chỉ có 3 đường được sử dụng :




Hình 9.9 : Giao tiếp với X2444

- P1.0 – SK ( serial clock : *clock* nối tiếp )
- P1.1 – CE ( chip enable : cho phép chip )
- P1.2 – DI/DO ( data input / output : dữ liệu nhập / xuất )

Các lệnh được chuyển đến X2444 bằng cách cho CE ở mức cao và kế đến dịch 8 bit của opcode bằng xung *clock* vào X2444 qua các đường SK và DI/DO. Các opcode sau đây cần cho thí dụ này :

| Lệnh | Opcode | Mô tả |
|---|---|---|
| RCL | 85H | Gọi lại dữ liệu trong EEROM vào RAM |
| WREN | 84H | Thiết lập chốt cho phép ghi |
| STORE | 81H | Lưu dữ liệu trong RAM vào EEROM |
| WRITE | 1AAAA011B | Ghi dữ liệu vào RAM ở địa chỉ AAAA |
| READ | 1AAAA111B | Đọc dữ liệu từ RAM ở địa chỉ AAAA |

**Mục tiêu thiết kế**

*Viết 2 chương trình. Chương trình thứ nhất, gọi là SAVE, sao chép nội dung của các vị trí nhớ nội từ 60H đến 7FH của 8051 vào EEROM của X2444. Chương trình thứ hai, gọi là RECOVER, đọc các dữ liệu đã lưu trước đây từ EEROM của X2444 và khôi phục chúng vào các vị trí nhớ từ 60H đến 7FH.*

Ta có 2 chương trình riêng biệt. Chương trình SAVE được thực thi mỗi khi thông tin trong NVROM được thay đổi ( thí dụ người sử dụng thay đổi các tham số cấu hình ). Chương trình RECOVER được thực thi mỗi khi hệ thống được cung cấp điện hoặc được *reset*. Với thí dụ này, thông tin được giữ trong các vị trí nhớ nội của 8051 từ địa chỉ 60H đến 7FH.

Liệt kê của phần mềm được trình bày ở hình 9.10.

Các thao tác lưu giữ và phục hồi dữ liệu bao gồm các bước sau :

**Ghi dữ liệu vào X2444**

1. Thực thi lệnh RCL
2. Thực thi lệnh WREN
3. Ghi dữ liệu vào RAM của X2444
4. Thực thi lệnh STO ( lưu RAM vào EEROM )
5. Thực thi lệnh SLEEP

**Đọc dữ liệu từ X2444**

1. Thực thi lệnh RCL
2. Đọc dữ liệu từ RAM của X2444
3. Thực thi lệnh SLEEP

Hình 9.11 minh họa giản đồ thời gian khi ta gởi lệnh RCL đến X2444. Một vài bit thực tế là tùy định nhưng được cho bằng 0 trong hình 9.11.

Việc định thời cho các lệnh WRITE và READ dữ liệu có hơi khác. Với các lệnh này, opcode 8-bit được theo ngay sau bởi 16 bit dữ liệu và CE được duy trì ở mức cao cho cả 24 bit. Với lệnh đọc, 8-bit của opcode được ghi đến X2444, kế đến 16 bit dữ liệu được đọc từ X2444. Các chương trình con riêng rẽ được sử dụng cho việc đọc 8 bit ( R_BYTE, các dòng 106 đến 112 ) và cho việc ghi 8 bit ( W_BYTE, các dòng từ 117 đến 123 ).

```
1   $DEBUG
2   $NOPAGING
3   $NOSYMBOLS
4   ; FILE : NVRAM.SRC
5   ;*****************************************
6   ;              X2444 INTERFACE EXAMPLE
7   ;
8   ; Hai chương trình con được trình bày dưới đây dùng để
9   ; SAVE và RECOVER dữ liệu giữa NVRAM X2444 và
```

Hình 9.10 : Phần mềm giao tiếp với X2444

```
; 32 byte RAM nội của 8051
;****************************************
RECALL      EQU     85H             ; lệnh " recall "
WRITE       EQU     84H             ; lệnh " write enable "
STORE       EQU     81H             ; lệnh " store "
SLEEP       EQU     82H             ; lệnh " sleep "
W_DATA      EQU     83H             ; lệnh " write data "
R_DATA      EQU     87H             ; lệnh " read data "
MON51       EQU     00BCH           ; điểm nhập của MON51
LENGTH      EQU     32              ; 32 byte được cất/khôi phục
DIN         BIT     P1.2            ; các đường giao tiếp
ENABLE      BIT     P1.1            ; với X2444
CLOCK       BIT     P1.0
            DSEG    AT      60H
NVRAM:      DS      LENGTH
            CSEG    AT      8000H
WX2444:     ACALL   SAVE
            LJMP    MON51
RX2444:     ACALL   RECOVER
            LJMP    MON51
;****************************************
;     SAVE 8031 RAM LOCATIONS 60H – 7FH IN X2444 NVRAM
;****************************************
SAVE:       MOV     R0, #NVRAM      ; R0 → các vị trí để cất
            CLR     ENABLE          ; vô hiệu X2444
            MOV     A, #RECALL      ; lệnh " recall "
            SETB    ENABLE
            ACALL   W_BYTE
```

Hình 9.10 : ( tiếp theo )

```
                CLR     ENABLE
                MOV     A, #WRITE        ; lệnh " write enable " chuẩn
                SETB    ENABLE           ; bị để ghi vào X2444
                ACALL   W_BYTE
                CLR     ENABLE
                MOV     R7, #0           ; R7 = địa chỉ của X2444
AGAIN:          MOV     A, R7            ; đặt địa chỉ vào ACC
                RL      A                ; đưa vào các bit 3, 4, 5, 6
                RL      A
                RL      A
                ORL     A, #W_DATA       ; thiết lập lệnh ghi
                SETB    ENABLE
                ACALL   W_BYTE
                MOV     R5, #2
LOOP:           MOV     A, @R0           ; lấy dữ liệu của 8051
                INC     R0               ; trỏ đến byte kế
                ACALL   W_BYTE           ; gởi byte đến X2444
                DJNZ,   R5, LOOP         ; lặp lại ( gởi byte thứ hai )
                CLR     ENABLE
                INC     R7               ; tăng địa chỉ của X2444
                CJNE    R7, #16, AGAIN; nếu chưa kết thúc : tiếp tục
                MOV     A, #STORE        ; nếu kết thúc, sao chép vào
                SETB    ENABLE           ; EEROM
                ACALL   W_BYTE
                CLR     ENABLE
                MOV     A, #SLEEP        ; cho X2444  nghỉ
                SETB    ENABLE
                ACALL   W_BYTE
                CLR     ENABLE
                RET
```

Hình 9.10 : ( tiếp theo )

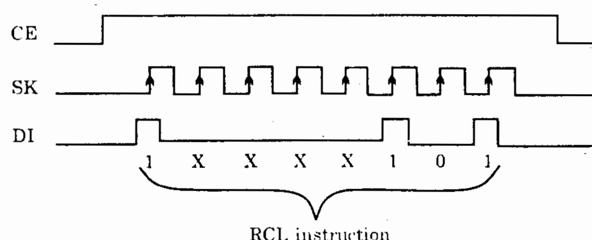
```
;****************************************
; RECOVER 8051 RAM LOCATIONS 60H – 7FH FROM X2444 NVRAM
;****************************************
RECOVER:   MOV     R0, #NVRAM
           CLR     ENABLE
           MOV     A, #RECALL      ; lệnh " recall "
           SETB    ENABLE
           ACALL   W_BYTE
           CLR     ENABLE
           MOV     R7, #0          ; R7 = địa chỉ của X2444
AGAIN2:    MOV     A, R7           ; đặt địa chỉ vào ACC
           RL      A               ; thiết lập lệnh đọc
           RL      A
           RL      A
           ORL     A, #R_DATA
           SETB    ENABLE
           ACALL   W_BYTE          ; gởi lệnh đọc
           MOV     R5, #2          ;  ( + địa chỉ )
LOOP2:     ACALL   R_BYTE          ; đọc byte dữ liệu
           MOV     @R0, A          ; đặt vào RAM của 8051
           INC     R0              ; trỏ tới vị trí kế
           DJNZ,   R5, LOOP2
           CLR     ENABLE
           INC     R7              ; tăng địa chỉ của X2444
           CJNE    R7, #16, AGAIN2; lặp lại
           MOV     A, #SLEEP       ; cho X2444  nghỉ
           SETB    ENABLE
           ACALL   W_BYTE
           CLR     ENABLE
           RET
```

Hình 9.10 : ( tiếp theo )

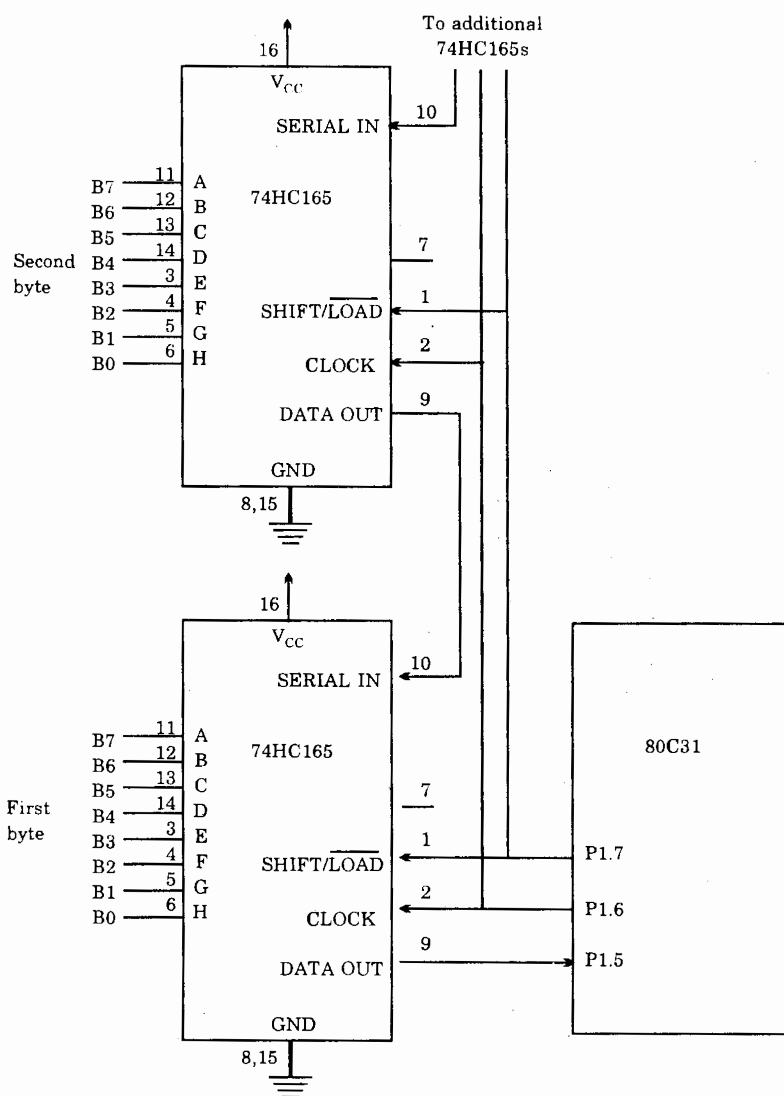
```
; ***************************************
; READ A BYTE OF DATA FROM X2444
; ***************************************
R_BYTE:     MOV     R6, #8          ; dùng R6 làm bộ đếm bit
AGAIN3:     MOV     C, DIN          ; đặt 1 bit của X2444 vào C
            RLC     A               ; thiết lập byte trong ACC
            SETB    CLOCK
            CLR     CLOCK
            DJNZ    R6, AGAIN3      ; nếu chưa là bit cuối, tiếp
            RET

; ***************************************
; WRITE A BYTE OF DATA TO X2444
; ***************************************
W_BYTE:     MOV     R6, #8          ; dùng R6 làm bộ đếm bit
AGAIN4:     RLC     A               ; đặt 1 bit vào C
            MOV     DIN, C          ; đặt lên đường DATA IN
            SETB    CLOCK
            CLR     CLOCK
            DJNZ    R6, AGAIN4      ; nếu không là bit cuối, tiếp
            RET
            END
```

Hình 9.10 : ( tiếp theo )




Hình 9.11 : Giản đồ thời gian cho lệnh " recall " RCL




Hình 9.12 : Giao tiếp với 2 IC 74HC165

## 9.7 MỞ RỘNG XUẤT NHẬP

Thí dụ kế tiếp minh họa một cách đơn giản biện pháp tăng số đường nhập của 8051. Các đường của 3 *port* được dùng để giao tiếp ( trong thí dụ ở mục này ) với nhiều 74HC165, các thanh ghi dịch nhập song song xuất nối tiếp ( xem hình 9.12 ). Các ngõ vào thêm vào được lấy mẫu tuần hoàn bằng cách cho đường SHIFT/LOAD xuống mức thấp. Kế đến dữ liệu được đọc vào 8051 bằng cách đọc đường DATA IN với xung *clock* dịch bit trên đường CLOCK. Mỗi một xung trên đường CLOCK dịch 1 bit dữ liệu, do vậy lần đọc kế ở DATA IN sẽ đọc bit kế và v.v... .

**Mục tiêu thiết kế**

*Viết một chương trình con sao chép trạng thái của 16 đường nhập trong hình 9.12 vào các vị trí nhớ 25 H và 26 H của RAM nội 8051.*

Phần mềm được trình bày trong hình 9.13. Lưu ý là vòng lặp của chương trình chính bao gồm các lời gọi hai chương trình con : GET_BYTES và DISPLAY_RESULTS ( các dòng 34, 35 ). Chương trình con sau minh họa một kỹ thuật thông dụng để gỡ rối khi các tài nguyên bị giới hạn. DISPLAT_RESULTS ( các dòng từ 72 đến 83 ) đọc dữ liệu từ các vị trí nhớ 25H và 26H của RAM nội và gởi từng ½ byte đến VDT dưới dạng ký tự số hex. Điều này cho ta sự giao tiếp đơn giản và trực quan để kiểm tra có phải chương trình và mạch giao tiếp đang làm việc. Khi các đường nhập chuyển trạng thái, sự thay đổi sẽ lập tức xuất hiện trên VDT.

Chương trình con GET_BYTES ( các dòng từ 44 đến 58 ) thực thi trong 112 µsec khi hai IC 74HC165 được sử dụng và hệ thống hoạt động ở tần số 12 MHz. Nếu các ngõ vào được lấy mẫu, thí dụ, 20 lần trong 1 sec, GET_BYTES sẽ tiêu phí mất 112 / 50000 = 0.2 % thời gian thực thi của CPU. Tuy nhiên điều này là tối thiểu; việc tăng số đường nhập và/hoặc tốc độ lấy mẫu có thể bắt đầu tác động lên hiệu suất của toàn hệ thống.

```
1   $DEBUG
2   $NOPAGING
3   $NOSYMBOLS
4   ; FILE : HC165.SRC
5   ;*****************************************
6   ;                  74HC165 INTERFACE EXAMPLE
7   ;
```

Hình 9.13 : Phần mềm giao tiếp với 74HC165

```
; Chương trình con GET_BYTES dưới đây đọc ( trong trường hợp này là
; 2 ) nhiều IC 74HC165, các thanh ghi dịch nhập song song xuất nối tiếp
; nối với chân P1.7 ( SHIFT/LOAD ), P1.6 ( CLOCK ) và P1.5 ( DATA-
; OUT ). Các byte đọc được sẽ được đặt vào các vị trí định địa chỉ bit
; có địa chỉ bắt đầu là BUFFER.
;
;*******************************************
CR          EQU     0DH
COUNT       EQU     2               ; số IC 74HC165
SHIFT       BIT     P1.7            ; ngõ vào SHIFT/LOAD
                                    ; 1 = dịch ; 0 = nạp
CLOCK       BIT     P1.6            ; ngõ vào xung clock
DOUT        BIT     P1.5            ; ngõ ra DATA OUT
OUTSTR      CODE    0282H           ; chương trình con của MON51

OUT2HEX     CODE    028DH           ; xuất byte dạng 2 số hex
OUTCHR      CODE    01DEH

            ORG     8000H           ; bắt đầu segment mã
            SETB    CLOCK           ; thiết lập các đường giao tiếp
            SETB    SHIFT
            SETB    DOUT

;*******************************************
; MAIN LOOP ( KEPT SMALL FOR THIS EXAMPLE )
;*******************************************
            CALL    SEND_HELLO_MESSAGE
REPEAT:     CALL    GET_BYTES       ; đọc 74HC165
            CALL    DISPLAY_RESULTS
            JMP     REPEAT          ; lặp vòng

```

Hình 9. 13 : ( tiếp theo )

```
;*****************************************
; GET BYTES FROM 74HC165 & PLACE IN INTERNAL RAM
;
; Thời gian thực thi = 112 µsec ( @ 12 MHz )
; Thời gian thực thi cho N 74HC165 = 6 / ( N x 53 ) µsec
;*****************************************
GET_BYTES:
             MOV     R6, #COUNT     ; dùng R6 làm bộ đếm byte
             MOV     R0. #BUFFER    ; dùng R0 làm con trỏ
             CLR     SHIFT          ; nạp vào 74HC165 bằng cách
             SETB    SHIFT          ; cho SHIFT/LOAD ở mức thấp
AGAIN:       MOV     R7, #8         ; dùng R7 làm bộ đếm bit
LOOP:        MOV     C, DOUT
             RRC     A
             CLR     CLOCK          ; dịch bit
             SETB    CLOCK
             DJNZ    R7, LOOP       ; không phải bit 8, tiếp
             MOV     @R0, A         ; là bit 8, cất vào bộ đệm
             INC     R0             ; tăng con trỏ
             DJNZ    R6, AGAIN      ; lấy 2 byte
             RET

;*****************************************
; READ HELLO MESSAGE TO CONSOLE ( DEBUGGING AID )
;*****************************************
SEND_HELLO_MESSAGE:
             MOV     DPTR, #BANNER
             CALL    OUTSTR
             RET
BANNER:      DB      ‘ *** TEST 74HC165 INTERFACE *** ‘, CR, 0

```

Hình 9.13 : ( tiếp theo )

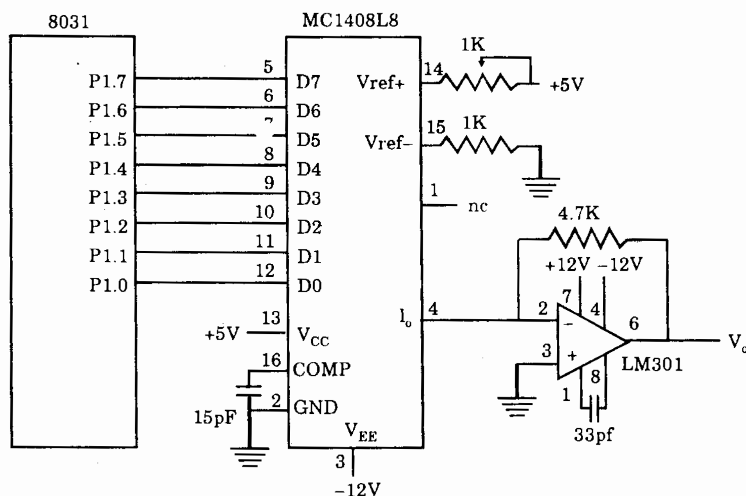
```
;***************************************************
; DISPLAY RESULTS ON CONSOLE ( DEBUGGING AID )
;***************************************************
DISPLAY_RESULTS:
                MOV     R0, #BUFFER   ; R0 trỏ tới các byte
                MOV     R6, #COUNT    ; R6 chứa số byte
LOOP2:          MOV     A, @R0        ; lấy byte
                INC     R0
                CALL    OUT2HEX
                MOV     A, #' '
                CALL    OUTCHR
                DJNZ    R6, LOOP2
                MOV     A, #CR
                CALL    OUTCHR
                RET

;***************************************************
; CREATE BUFFER IN BIT-ADDRESSABLE INTERNAL RAM
;***************************************************
                DSEG    AT      25H
BUFFER:         DS      COUNT
                END
```

Hình 9.13 : ( tiếp theo )

## 9.8 XUẤT TÍN HIỆU TƯƠNG TỰ

Việc giao tiếp với thế giới thực thường yêu cầu tạo ra các tín hiệu tương tự ( analog ). Việc tạo và điều khiển một tín hiệu xuất dạng tương tự từ một bộ vi xử lý không có gì khó khăn. Thí dụ này sử dụng hai điện trở, hai tụ điện, một biến trở, một opamp LM301 và một bộ biến đổi số sang tương tự DAC 8-bit MC1408L8. Cả hai IC đều không đắt. 8 ngõ vào dữ liệu của bộ biến đổi DAC được nối với *port* 1 trên 8031 ( xem hình 9.14 ). Sau khi thiết lập mạch và kết nối với SBC-51, mạch cần được kiểm tra bằng cách sử dụng các lệnh điều khiển của chương trình *monitor*. Ta đo điện áp ngõ ra ở chân 6 của LM301 trong khi đang ghi các giá trị khác cho *port* 1 và đang hiệu chỉnh biến trở 1 K. Ngõ ra phải thay đổi từ 0 V ( P1 = 00H ) cho đến khoảng 10 V ( P1 = FFH ).




Hình 9.14 : Giao tiếp với DAC

Sau khi mạch đã hoạt động đúng, ta sẽ cho phần mềm hoạt động. Chương trình kiểm tra thông dụng là chương trình tạo sóng tam giác. Chương trình này gởi một giá trị đến DAC, tăng giá trị này rồi gởi tiếp, v.v... Tuy nhiên, chúng ta sẽ tiến hành trên một thiết kế nhiều tham vọng hơn – mạch tạo sóng sin từ tín hiệu số.

**Mục tiêu thiết kế**

*Viết một chương trình được điều khiển ngắt, tạo ra sóng sin bằng cách sử dụng giao tiếp với DAC trong hình 9.14. Ta sử dụng một hằng số gọi là STEP để thiết lập tần số của sóng sin.*

Vì khả năng xử lý các con số của 8031 có nhiều giới hạn, chỉ có một phương pháp hợp lý cho vấn đề này là sử dụng bảng tìm kiếm. Chúng ta cần một bảng có các giá trị 8-bit tương ứng với một chu kỳ của sóng sin. Các giá trị nên bắt đầu từ 127, tăng đến 255 và giảm từ 127 xuống 0 rồi lại tăng trở lại đến 127 theo biểu đồ của hình sin.

Một thể hiện hợp lý của sóng sin yêu cầu một bảng tương đối lớn nên vấn đề đặt ra là làm cách nào ta tạo được bảng ? Các phương pháp bằng tay là không thực tế. Phương pháp dễ dàng nhất là viết một chương trình bằng một ngôn ngữ cấp cao nào đó để tạo ra bảng và lưu các điểm nhập của tập tin. Sau đó bảng này được mang vào trong chương trình nguồn viết cho 8031.

Hình 9.15 là một chương trình C đơn giản có tên là table51.c. Chương trình này tạo ra một bảng sóng sin có 1024 điểm nhập với các giá trị nằm trong khoảng từ 0 đến 255. Kết quả tạo ra được ghi vào một

tập tin kết quả có tên là sine51.src. Mỗi một điểm nhập được đứng trước bởi chỉ dẫn DB để tương thích với mã nguồn của 8031.

```
/*************************************/
/* table51.c – chương trình tạo ra một bảng sóng sin       */
/*                                                         */
/* Bảng này bao gồm 1024 điểm nhập ở giữa 0 và 255.        */
/* Mỗi một điểm nhập được đứng trước bởi ' DB ' để tương   */
/* thích với chương trình nguồn của 8051. Bảng được ghi vào */
/* tập tin có tên là sine51.src                            */

#include <stdio.h>
#include <math.h>
#define PI      3.1415927
#define MAX     1024
#define BYTE    255
main()
(
    FILE *fp, *fopen();
    double x, y
    fp = fopen( "sine51.src", "w" );
    for (x = 0; x < MAX; ++x) (
        y = (( sin (( x/MAX ) * ( 2 * PI )) + 1 ) / 2 ) * BYTE;
        fprintf (fp, " DB %3d\n ", (int)y );
    )
)
```

Hình 9.15

Chương trình tạo sóng sin được trình bày ở hình 9.16. Vòng lặp chính ( các dòng từ 36 đến 40 ) thực hiện 3 công việc :

- khởi động bộ định thời 0 để tạo ngắt sau mỗi 100 μsec
- cho phép các ngắt

- đặt vào trong vòng lặp vô tận.

Trình phục vụ ngắt cho bộ định thời 0 ( các dòng từ 41 đến 51 ) thực hiện mọi công việc. Cứ mỗi một 100 μsec, một giá trị được đọc từ bảng tìm kiếm bằng cách sử dụng con trỏ DPTR và kế đến, giá trị này được ghi lên *port* 1. Một hằng số gọi là STEP được sử dụng để tăng. STEP được định nghĩa ở dòng 26 ở dạng byte trong RAM nội. STEP phải được khởi động bằng lệnh điều khiển của chương trình *monitor*. Trong mỗi một trình phục vụ ngắt, STEP được cộng với DPTR để gán địa chỉ cho mẫu kế. Bảng được bắt đầu ( bằng chỉ dẫn ORG ) ở địa chỉ 8400H ( dòng 69 ), như vậy bảng bắt đầu ở địa chỉ chẵn và là địa chỉ bắt đầu 1 K byte bộ nhớ. Nếu DPTR được tăng qua 87FFH ( điểm kết thúc bảng ), DPTR được điều chỉnh để xoay vòng về điểm bắt đầu bảng. Do bảng khá lớn, chỉ dẫn $NOLIST của trình dịch hợp ngữ đã được sử dụng sau 5 điểm nhập đầu tiên ( dòng 77 ) để không cho xuất hiện trong tập tin liệt kê. Chỉ dẫn $LIST được sử dụng ở dòng 1092 ( không được trình bày ) sẽ đưa danh sách trở về 5 điểm nhập sau cùng. Tần số của sóng sin được điều khiển bởi 3 tham số : STEP, kích thước của bảng và chu kỳ ngắt do bộ định thời, được giải thích ở các dòng từ 16 đến 20 trong bảng liệt kê.

```
1   $DEBUG
2   $NOPAGING
3   $NOSYMBOLS
4   ; FILE : DAC.SRC
5   ;********************************************
6   ;                MC1408L8 INTERFACE EXAMPLE
7   ;
8   ; Chương trình này tạo ra một sóng sin bằng cách dùng một bảng tìm
9   ; kiếm sóng sin và một giao tiếp với MC1408L8, bộ biến đổi số sang
10  ; tương tự DAC 8-bit.
11  ; Chương trình được điều khiển ngắt
12  ;
13  ; Dữ liệu được đọc từ một bảng sóng sin 1024 điểm nhập và gởi đến
14  ; DAC mỗi một 100 μsec.
15  ; Mỗi một giá trị được gởi là các vị trí STEP
16  ; qua vị trí trước đã được gởi ( và được vòng lại khi đã đạt đến cuối
17  ; bảng ).
```

Hình 9.16 : Phần mềm giao tiếp với DAC

```
; Chu kỳ của sóng sin là 100 x ( 1024 / STEP ) µsec. Thí dụ
; nếu STEP là 20H, sóng sin có chu kỳ là 100 x ( 1024 / 32 ) = 3.2 ms
; và tần số là 313 Hz
;
; Lưu ý : Ta khởi động STEP trong vị trí nhớ nội 50H
; trước khi chạy chương trình
;*****************************************
MONITOR     CODE    00BCH
STEP        DATA    50H             ; đặt STEP trong RAM nội

            ORG     8000H
            LJMP    MAIN            ; khởi động bộ định thời
            LJMP    EXT0ISR         ; không sử dụng
            LJMP    T0ISR           ; cập nhật DAC mỗi 100 µsec
            LJMP    EXT1ISR         ; không sử dụng
            LJMP    T1ISR           ; không sử dụng
            LJMP    SPISR           ; không sử dụng
            LJMP    T2ISR           ; không sử dụng
 MAIN:      MOV     TMOD, #02H      ; chế độ tự nạp lại 8-bit
            MOV     TH0, #-100      ; trì hoãn 100 µsec
            SETB    TR0             ; bắt đầu bộ định thời
            MOV     TE, #82H        ; cho phép ngắt do timer 0
            SJMP    $               ; vòng lặp chính không làm gì
T0ISR:      MOV     A, STEP         ; cộng STEP với DPTR
            ADD     A, DPL
            MOV     DPL, A
            JNC     SKIP
            INC     DPH
SKIP:       ANL     DPH, #03H       ; xoay vòng nếu cần
            ORL     DPH, #HIGH (TABLE)
            CLR     A
```

Hình 9.16 : ( tiếp theo )

```
                MOVC    A, @A+DPTR      ; lấy điểm nhập
                MOV     P1, A           ; gởi đi
                RETI

EXT0ISR
EXT1ISR
T1ISR
T2ISR
SPISR:          CLR     EA              ; cấm các ngắt và
                LJMP    MONITOR         ; trở về MON51

;****************************************
; Sau đây là bảng tìm kiếm sóng sin. Bảng này chứa 1024 điểm nhập
; và được bắt đầu ở địa chỉ 8400H để cho phép xoay vòng nội dung của
; DPTR mỗi khi đạt đến cuối bảng.
; Các điểm nhập là các điểm nhập 8-bit ( 0 đến 255 ).
; Bảng được tạo ra từ một chương trình C và
; được đặt vào trong chương trình viết cho 8051
;
;****************************************
                ORG     8400H
TABLE:          DB      127
                DB      128
                DB      129
                DB      129
                DB      130
; Bảng liệt kê ngưng sau 5 điểm nhập đầu tiên
;****************************************
$NOLIST
; Bảng liệt kê tiếp tục cho 5 điểmnhập sau cùng
                DB      123
                DB      124
```

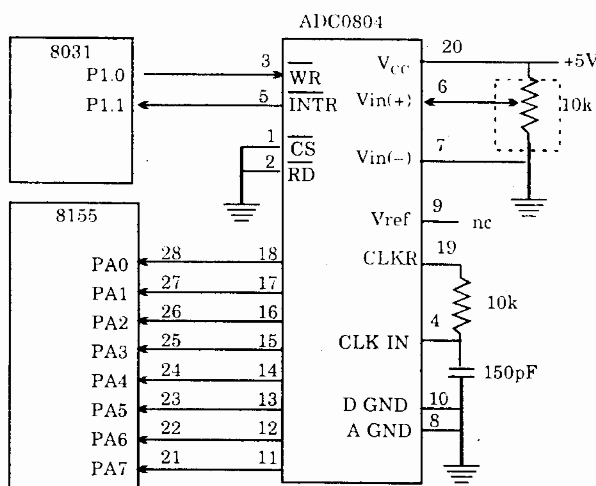
Hình 9.16 : ( tiếp theo )

```
        DB      125
        DB      125
        DB      126
        END
```

Hình 9.16 : ( tiếp theo )

## 9.9 NHẬP TÍN HIỆU TƯƠNG TỰ

Thí dụ thiết kế sau cùng là kênh nhập tín hiệu tương tự. Mạch ở hình 9.17 sử dụng 1 điện trở, 1 tụ điện, một biến trở vi chỉnh và 1 bộ biến đổi tương tự sang số ADC0804. ADC0804 là một bộ ADC không đắt, biến đổi điện áp tương tự thành tín hiệu số 8-bit trong khoảng 100 μsec.




Hình 9.17 : Giao tiếp với ADC

ADC0804 được điều khiển bởi một ngõ vào ghi $\overline{WR}$ và một ngõ ra ngắt $\overline{INTR}$. Việc biến đổi được bắt đầu bằng cách cho $\overline{WR}$ xuống mức thấp. Khi việc biến đổi hoàn tất ( 100 μsec sau ), ADC0804 xác lập $\overline{INTR}$ xuống mức thấp. $\overline{INTR}$ không được xác lập ( ở mức cao ) khi có chuyển trạng thái từ 1 → 0 kế của $\overline{WR}$, bắt đầu lần biến đổi ADC kế. $\overline{INTR}$ và $\overline{WR}$ nối với 8031 trên các đường P1.1 và P1.0. Với thí dụ này, chúng ta sử dụng *port* A của 8155 để truyền dữ liệu như được vẽ trên hình.

ADC0804 hoạt động nhờ vào một nguồn xung *clock* bên ngoài được tạo ra bằng cách nối mạch R-C với các chân 19 và 4. Điện áp ngõ vào

tương tự là tín hiệu sai biệt đặt trên các ngõ vào Vin(+) và Vin(-) trên các chân 6 và 7. Trong thí dụ này, Vin(-) nối đất ( 0 V ) và Vin(+) có được bằng cách điều chỉnh một biến trở vi chỉnh để thay đổi điện áp điểm giữa của biến trở. Vin(+) ở trong tầm từ 0 đến 5 V.

**Mục tiêu thiết kế**

*Viết một chương trình biến đổi điện áp tại điểm giữa của biến trở vi chỉnh thành tín hiệu số. Kết quả được hiển thị trên VDT dưới dạng mã ASCII.*

Chương trình được trình bày ở hình 9.18. Vì 8155 được mặc định là nhập sau khi *reset*, chuỗi khởi động IC này không cần thiết. *Port* A có địa chỉ là 0101H của bộ nhớ ngoài và được đọc dễ dàng bằng cách dùng lệnh MOVX. Việc biến đổi được bắt đầu bằng cách xóa và *set* P1.0 ( các dòng 34, 35 ), ngõ vào $\overline{WR}$ của ADC0804. Kế đến, chương trình ở trong vòng lặp chờ cho đến khi việc biến đổi kết thúc và xác lập $\overline{INTR}$ ở chân P1.1 ( dòng 36 ). Dữ liệu được đọc ở các dòng 37 và 38 và rồi được gởi đến VDT bằng cách dùng chương trình con OUT2HX ( dòng 39 ). Khi chương trình hoạt động, một byte được hiển thị trên VDT. Byte này ở trong tầm từ 00H đến FFH khi ta điều chỉnh biến trở vi chỉnh.

Chương trình ở hình 9.18 là chương trình tính xấp xỉ điện áp tương tự ở ngõ vào . Ta có thể thay thế biến trở vi chỉnh bởi các ngõ vào tương tự khác. Nhiệt độ nhận được bằng cách sử dụng một điện trở nhiệt – linh kiện có điện trở thay đổi theo nhiệt độ. Ta cũng có thể nhận tiếng nói bằng cách sử dụng một máy vi âm.

Chu kỳ biến đổi của ADC0804 là 100 μsec có được nhờ vào một tần số lấy mẫu bằng 10 KHz. Với tần số này, tín hiệu nhận được có băng thông lên đến 5 KHz, tương đương với băng thông của một đường truyền tín hiệu thoại. Một mạch khuếch đại được cần đến để khuếch đại mức tín hiệu thấp từ máy vi âm sao cho đạt được tầm 0 đến 5 V theo yêu cầu của 0804. Ngoài ra ta còn phải sử dụng một mạch lấy mẫu và giữ ( sample-and-hold circuit ) để duy trì một điện áp không đổi trong suốt thời gian của mỗi một chu kỳ biến đổi ADC.

```
1   $DEBUG
2   $NOPAGING
3   $NOSYMBOLS
4   ; FILE : ADC.SRC
5   ;***********************************
6   ;            ADC0804 INTERFACE EXAMPLE
7   ;
```

```
; Chương trình này đọc dữ liệu tương tự từ một
; ADC0804 giao tiếp với port A của 8155. Kết quả
; được cho hiển thị lên VDT dưới dạng một byte số hex
; Các bước như sau :
;
;           1. Gởi thông báo đến VDT
;           2. Xác lập WR ( P1.0 ) để bắt đầu biến đổi
;           3. Chờ INTR ( P1.1 ) chuyển thành mức thấp chỉ ra rằng đã kết
;              thúc biến đổi ADC
;           4. Đọc dữ liệu từ port A của 8155
;           5. Xuất dữ liệu đến VDT
;           6. Trở lại bước 2
;
;***************************************
PORTA       EQU     0101H           ; port A của 8155
CR          EQU     0DH             ; mã ASCII của các ký tự
LF          EQU     0AH
ESC         EQU     1BH
OUT2HX      EQU     028DH           ; chương trình con của MON51
OUTSTR      EQU     0282H           ; chương trình con của MON51
WRITE       BIT     P1.0            ; đường WR của ADC0804
INTR        BIT     P1.1            ; đường INTR của 0804

            ORG     8000H
ADC:        MOV     DPTR, #BANNER
            CALL    OUTSTR          ; gởi thông báo
LOOP:       CLR     WRITE           ; xác lập WR
            SETB    WRITE
            JB      INTR, $         ; chờ INTR = 0
            MOV     DPTR, #PORTA    ; khởi động DPTR → port A
            MOVX    A, @DPTR        ; đọc dữ liệu của ADC0804
            CALL    OUT2HX          ; gởi ra VDT
            MOV     DPTR, #LEFT2    ; cập nhật cursor bởi 2
            CALL    OUTSTR
            SJMP    LOOP            ; lặp lại

BANNER:     DB      '*** TEST ADC0804 ***', CR, 0
LEFT2:      DB      ESC, '(2D', 0   ; chuỗi thoát
            END
```

Hình 9. 18 : Phần mềm giao tiếp với ADC

# MỤC LỤC

**Phụ lục H**